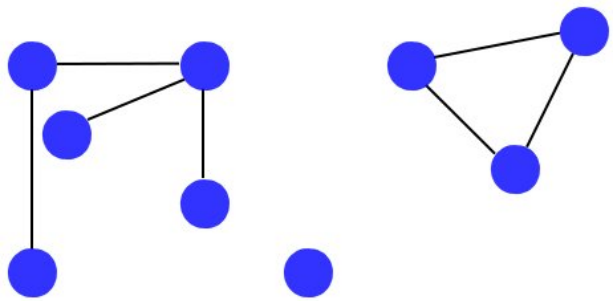
Graphs

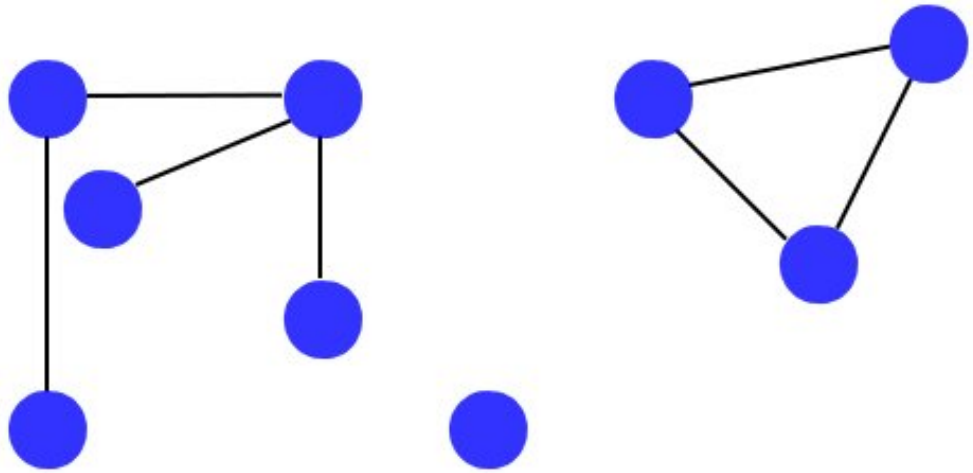
Dr Timothy Kimber

February 2015



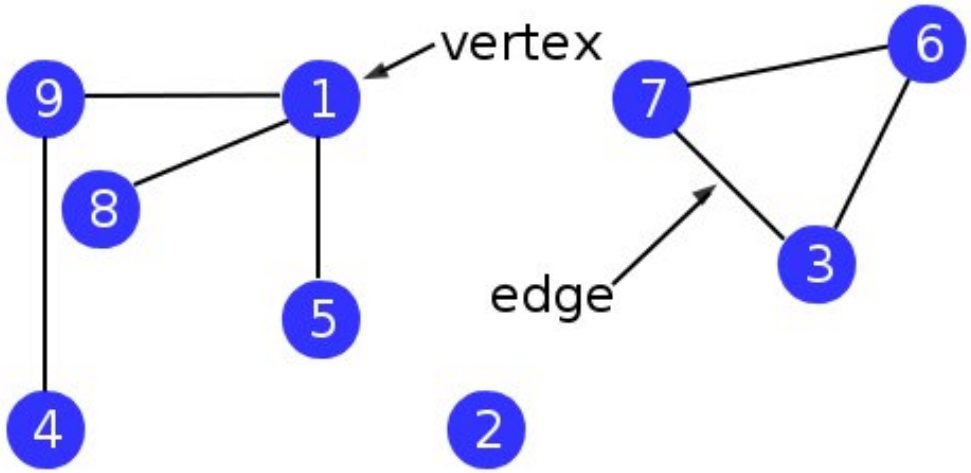
Introduction

Graphs are fundamental to much of computer science



- We have already seen how trees are used as data structures
- All sorts of problems can be modelled using graphs
- Networks, images, programs, anything involving related objects

Graph Terminology

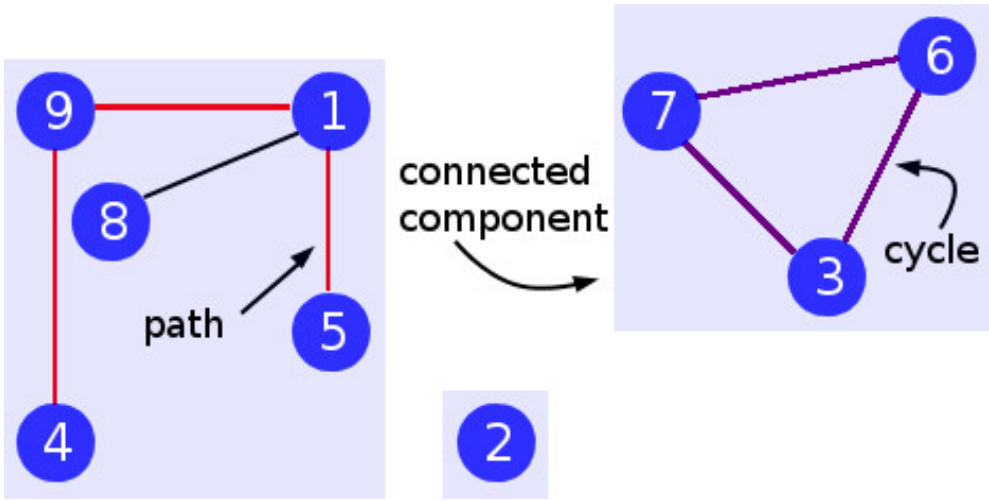


Definition

A graph G is a pair (V, E) where V is a finite set (of objects) and E is a binary relation on V. Elements of V are called vertices and elements of E are called edges.

- E is a set of pairs of vertices: $\{u, v\}$ such that there is an edge between u and v
- Vertices u and v are adjacent if there is an edge $\{u, v\}$

Graph Terminology



- A path from v_1 to v_n, written $v_1 \rightsquigarrow v_n$, is a sequence $\langle v_1, v_2, \ldots, v_n \rangle$ such that there is an edge $\{v_i, v_{i+1}\}$ for all i, $1 \leq i < n$
- A cycle exists if there is a path from v to v, for some vertex v
- Vertex v is reachable from vertex u if $u = v$, or if there is a path $u \rightsquigarrow v$
- A connected component (also just called a component) is a set of vertices all reachable from each other

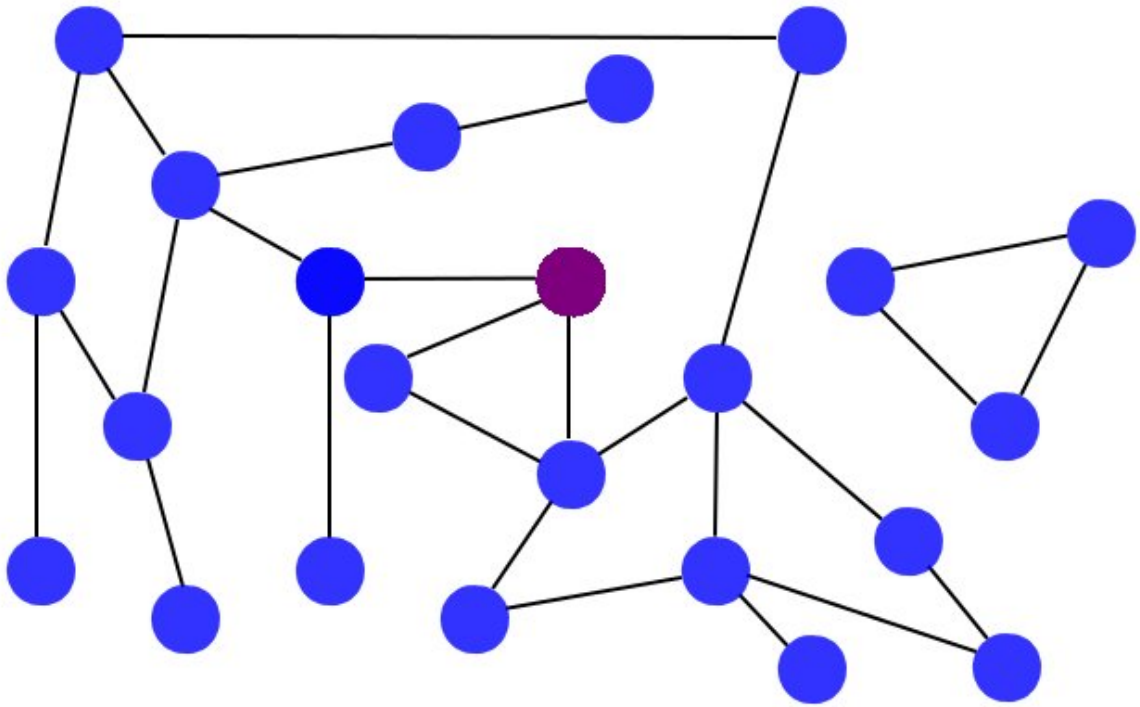
Part I

Breadth-First Search

Breadth-First Search

A graph has to be traversed in order to determine its properties

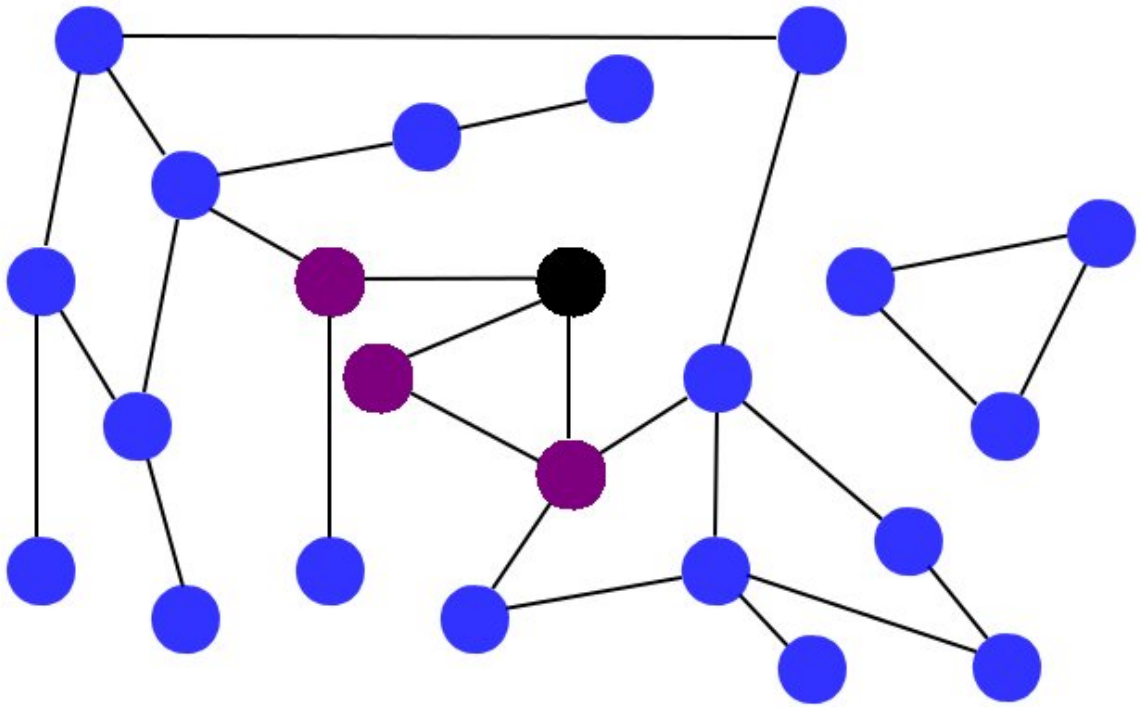
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

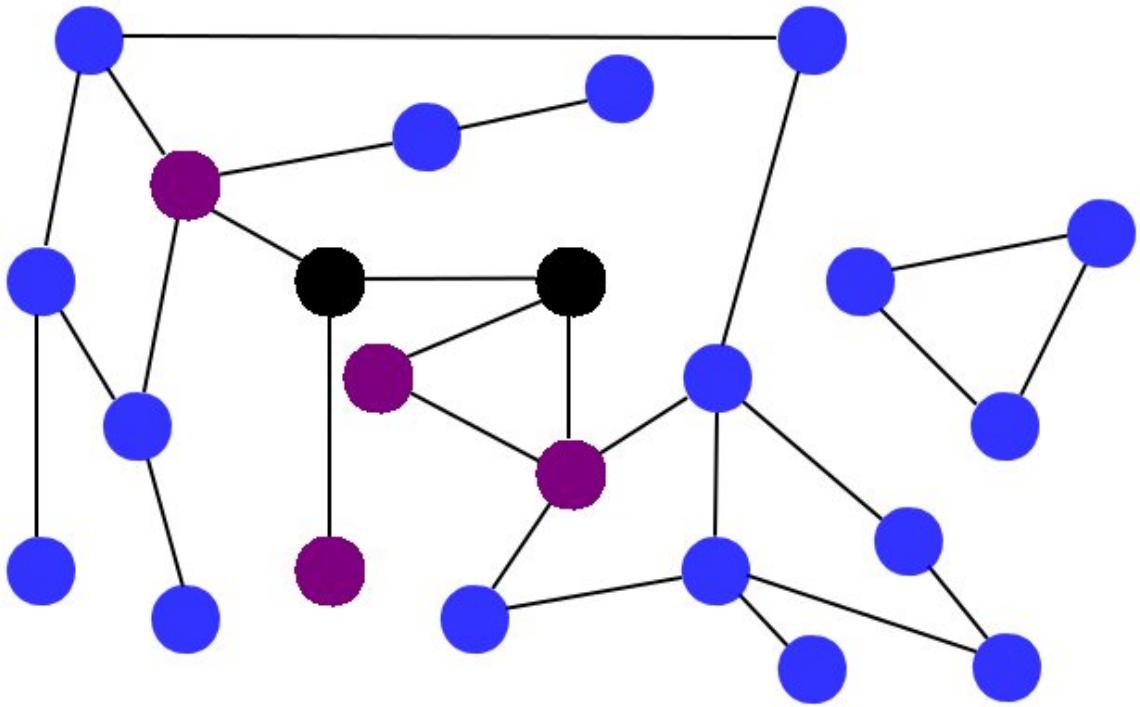
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

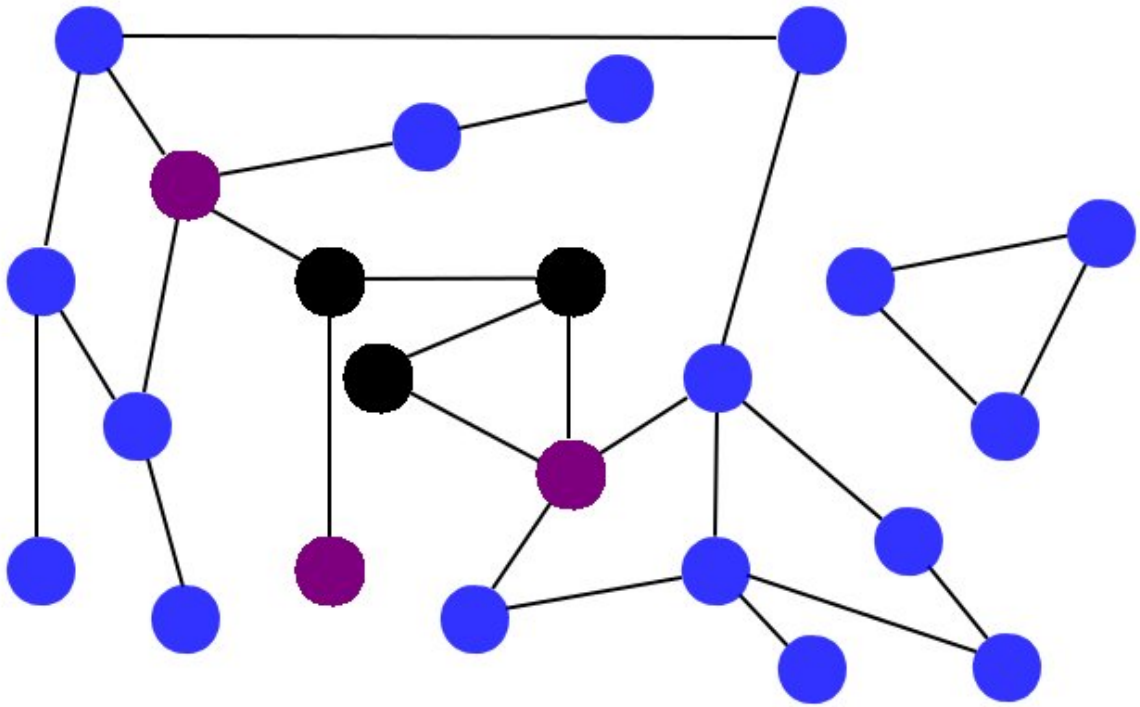
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

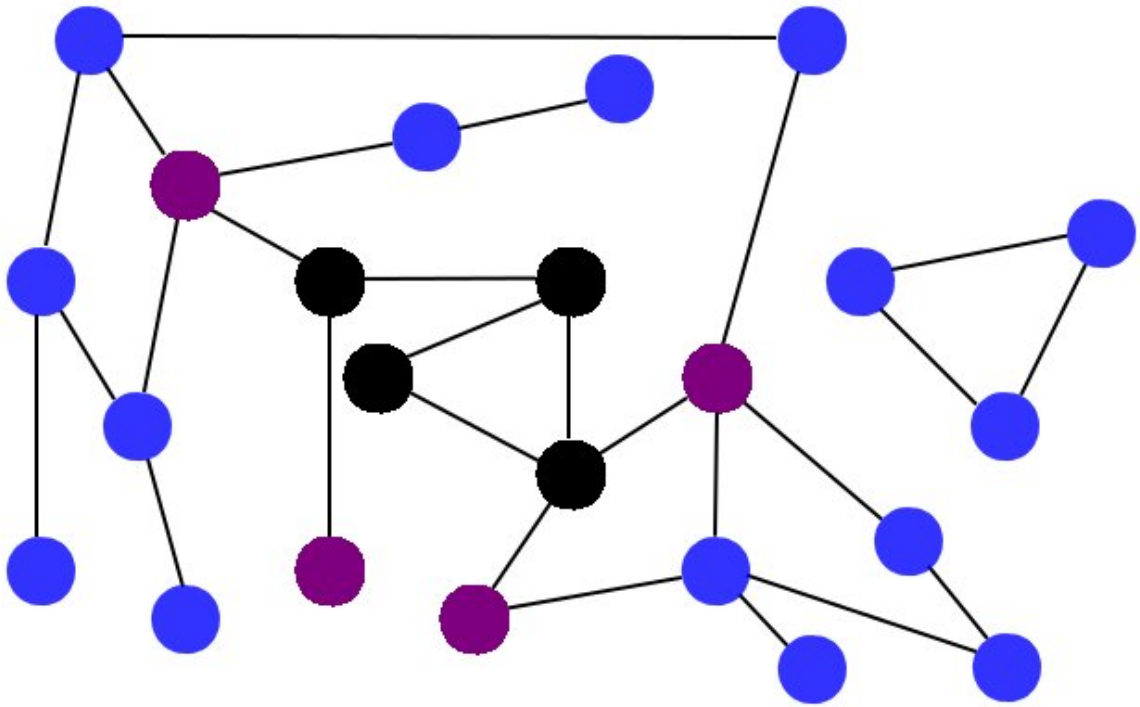
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

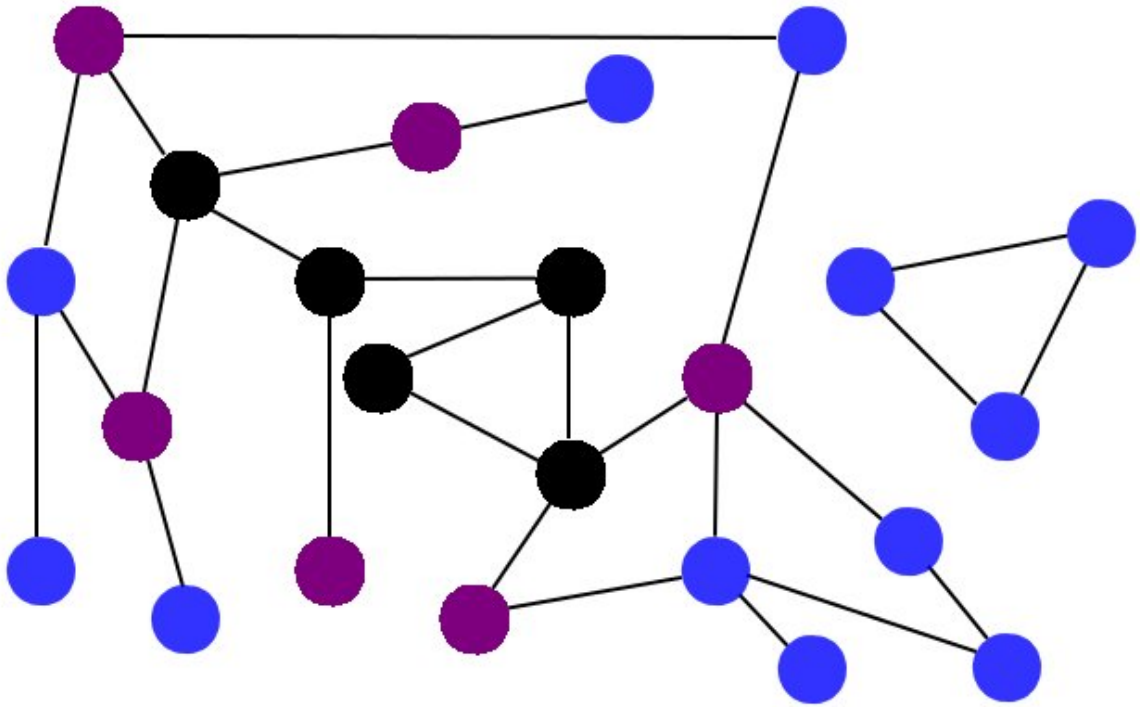
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

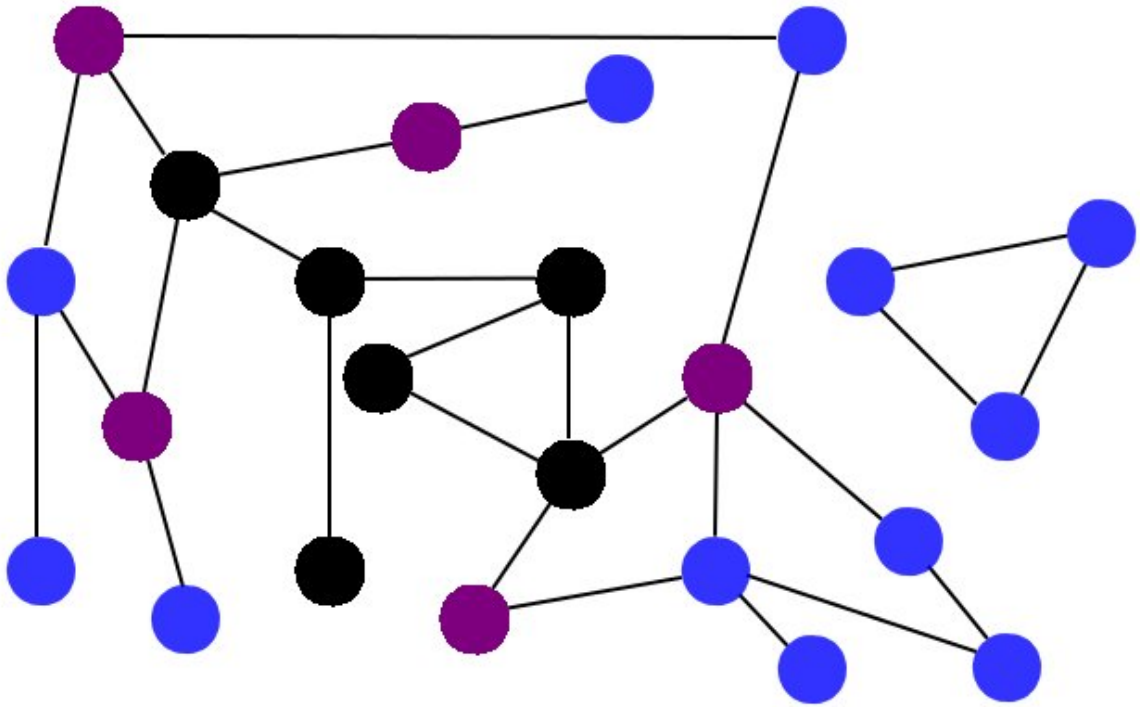
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

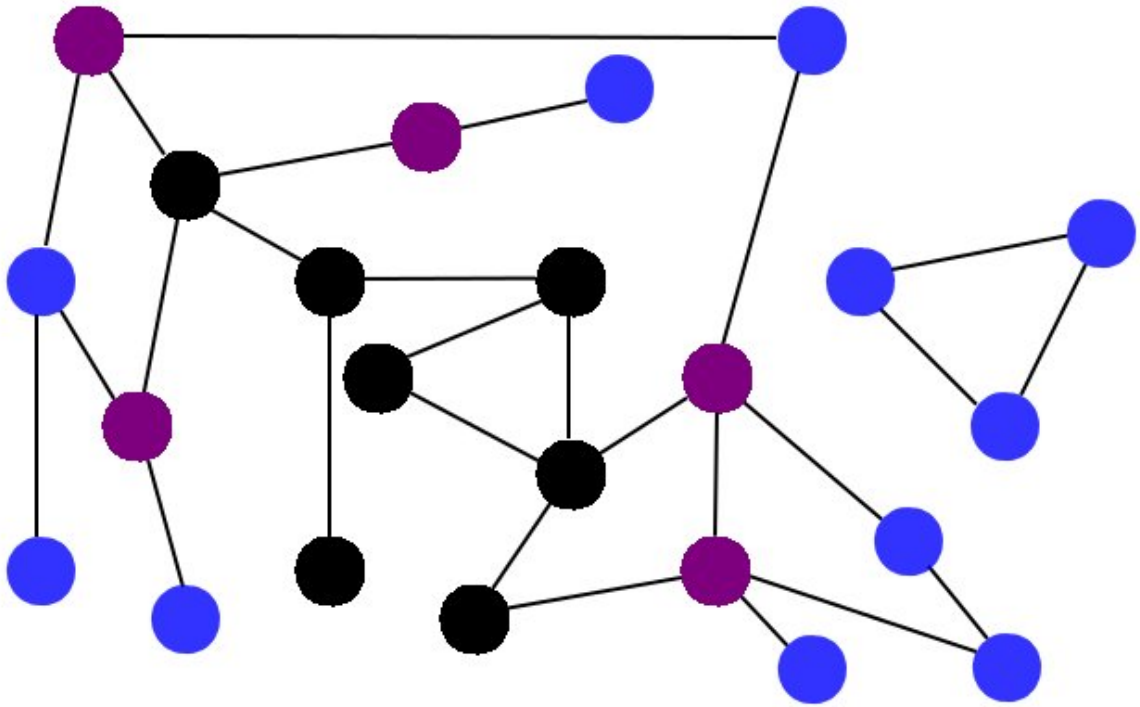
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

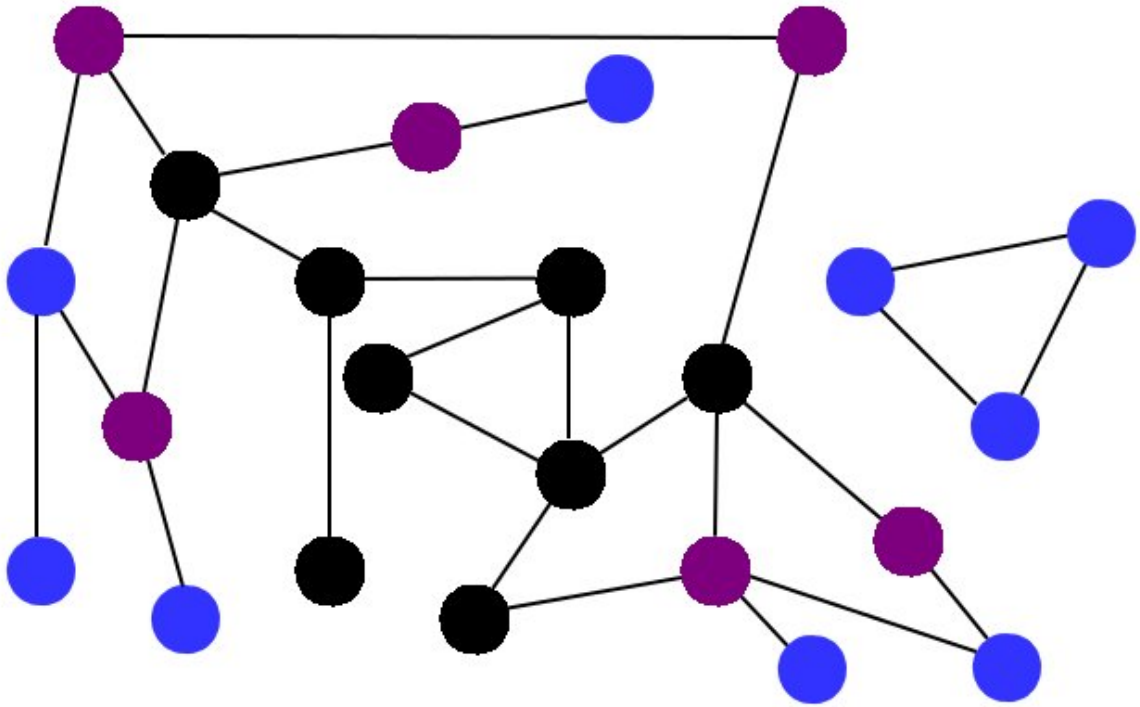
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

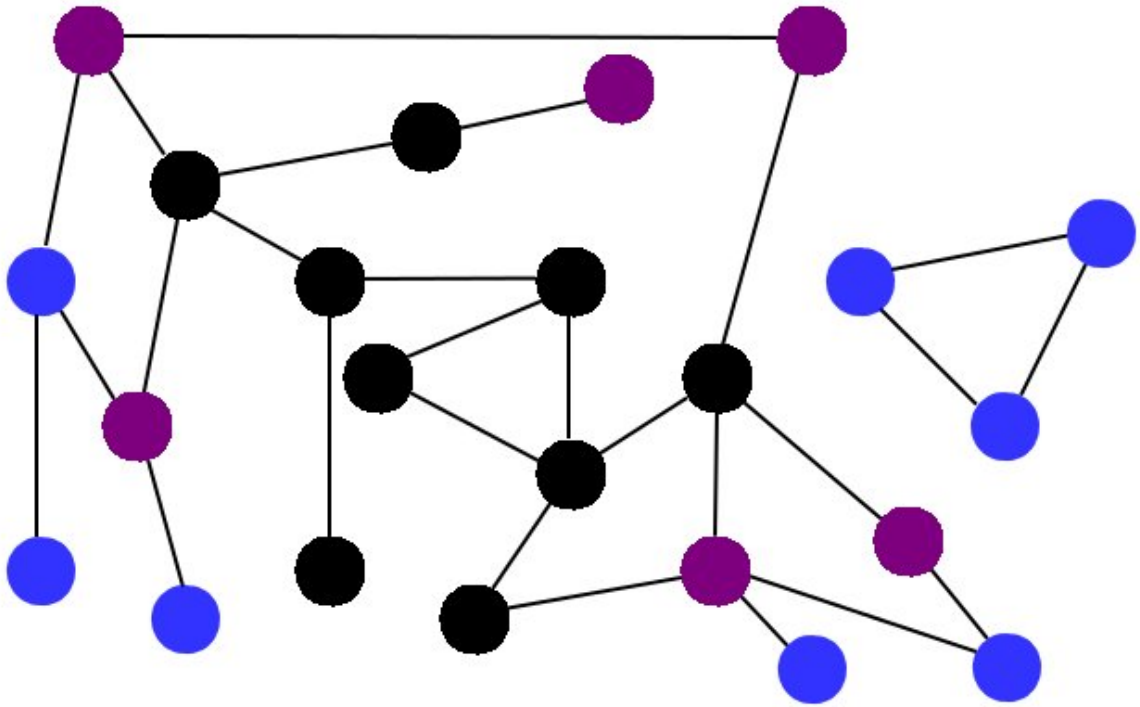
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

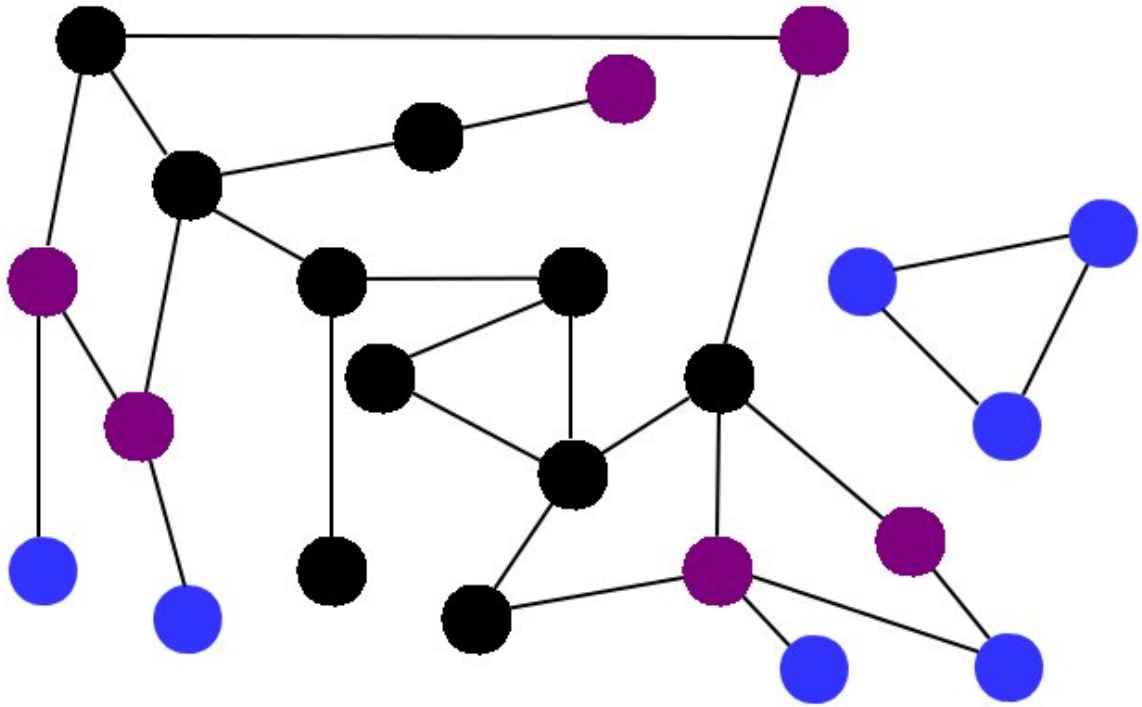
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

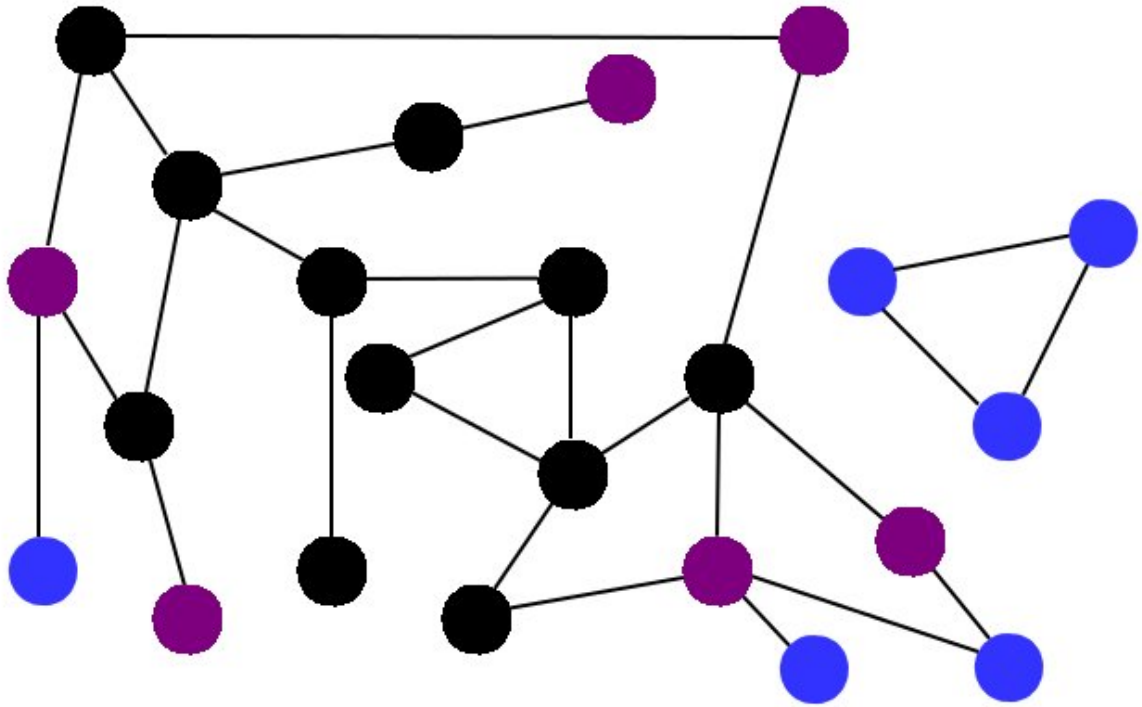
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

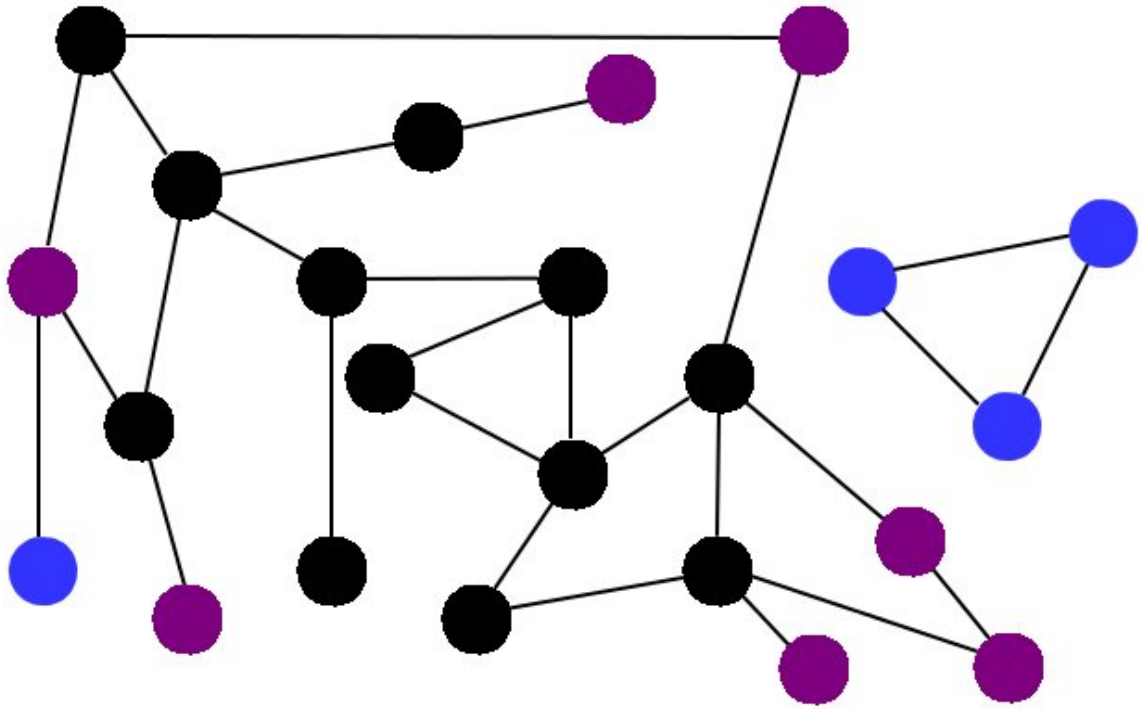
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

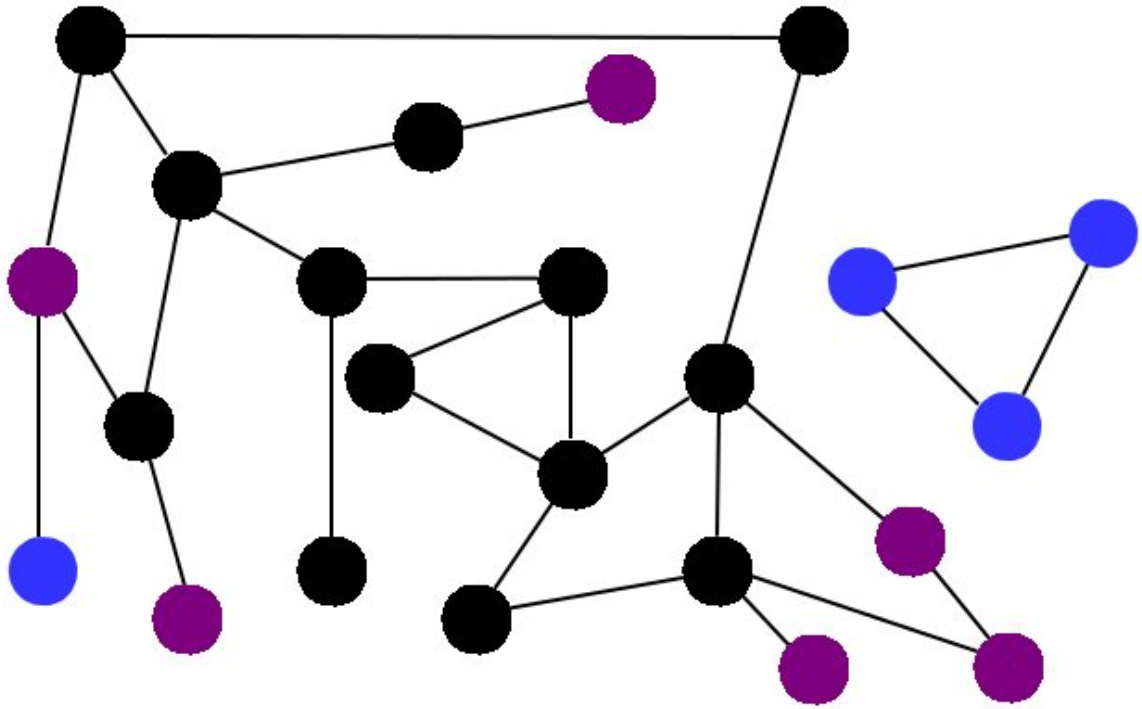
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

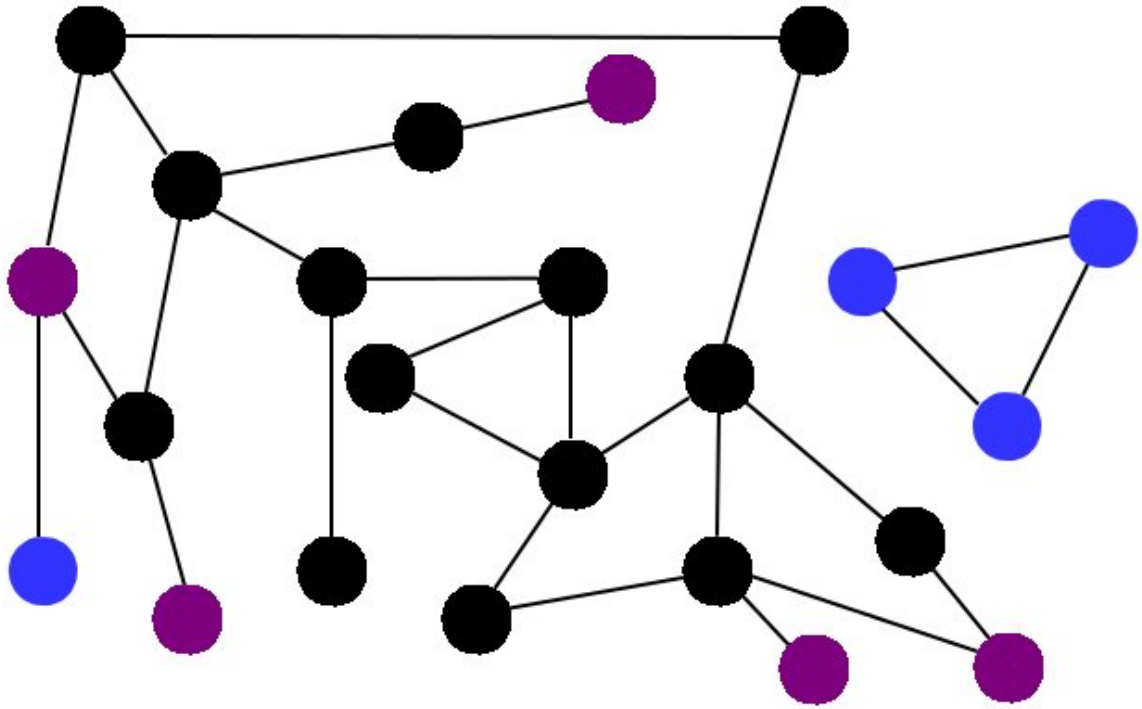
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

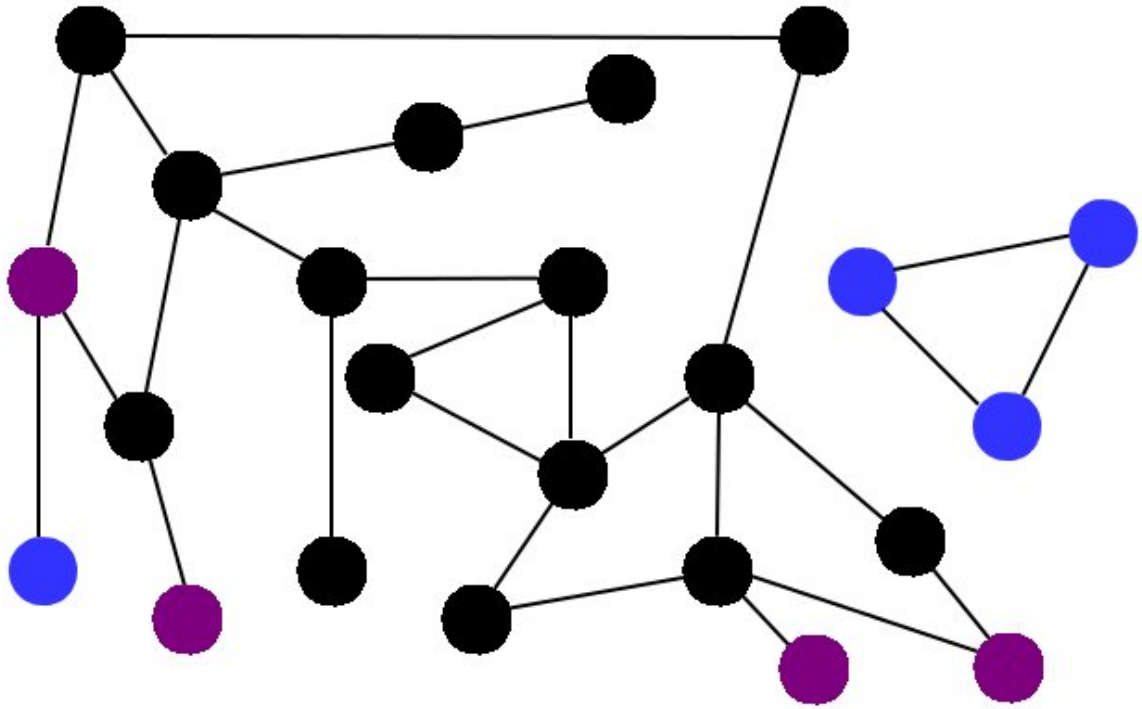
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

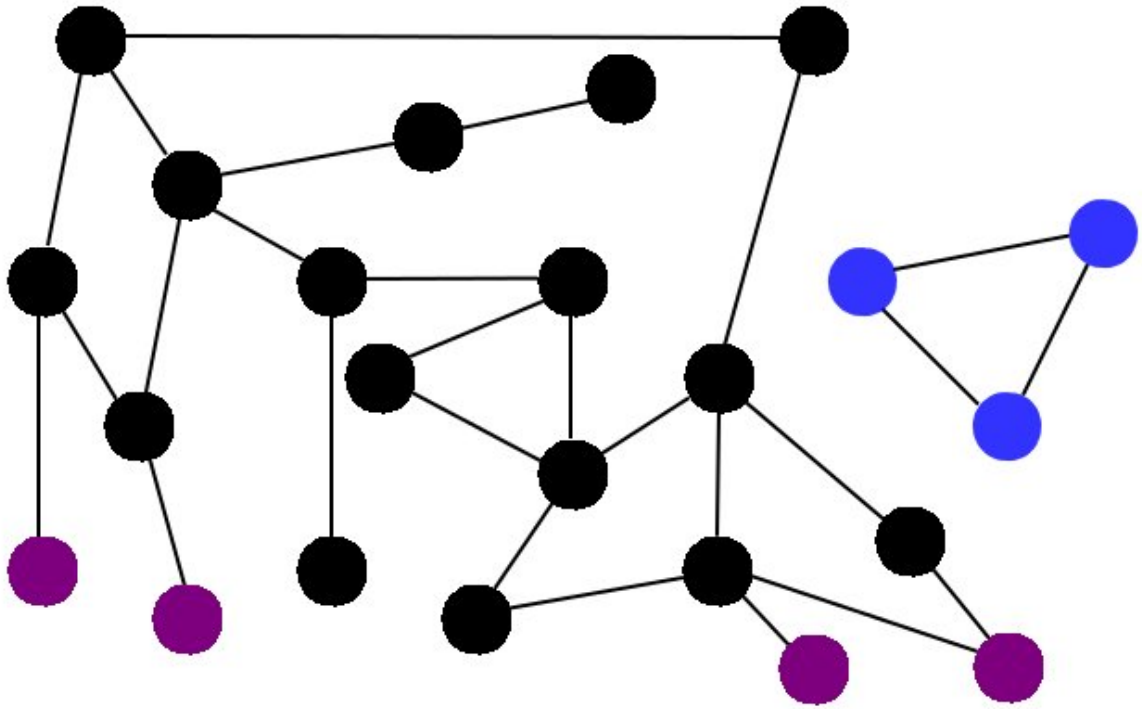
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

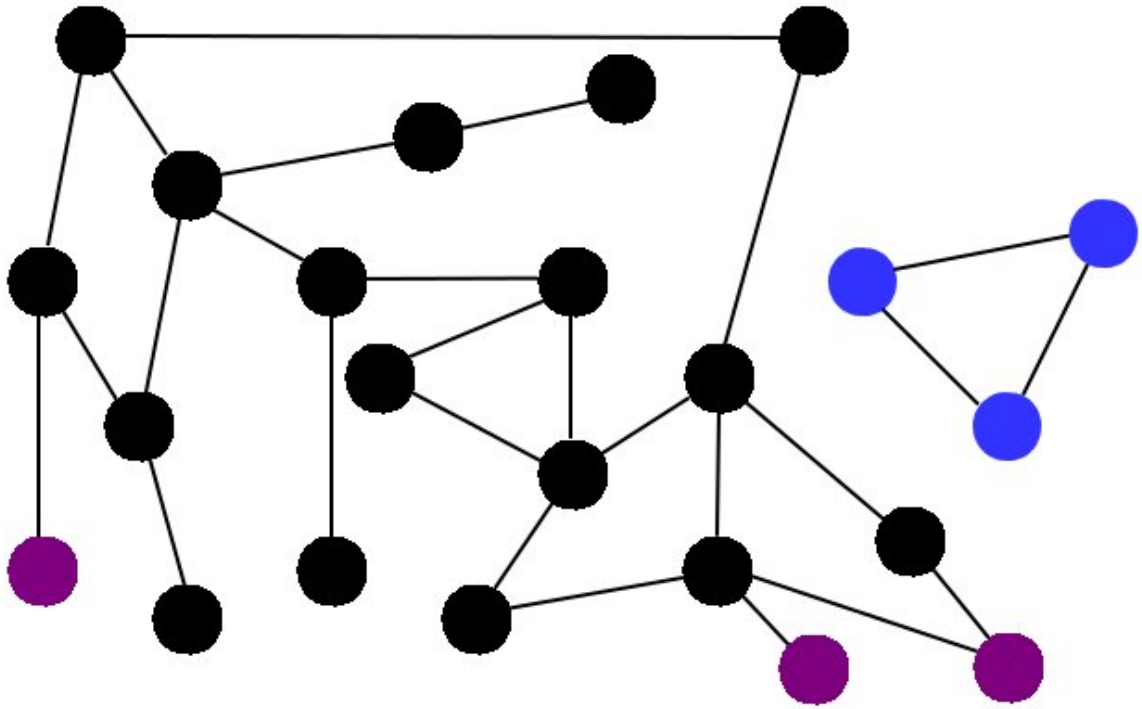
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

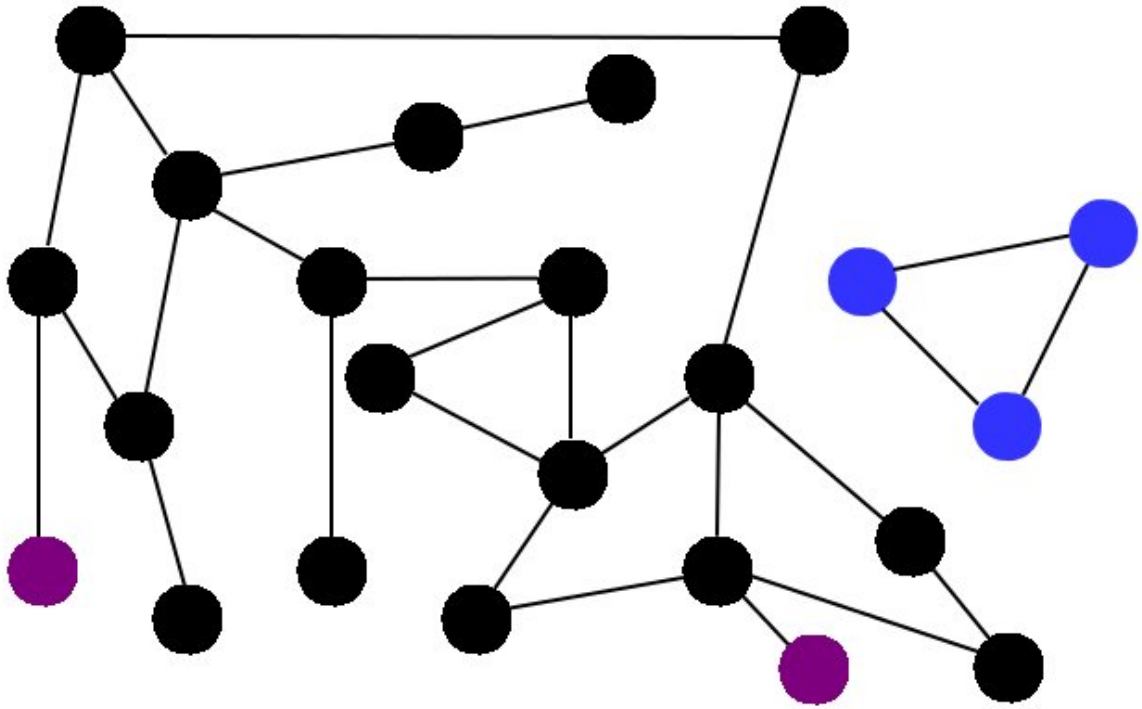
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

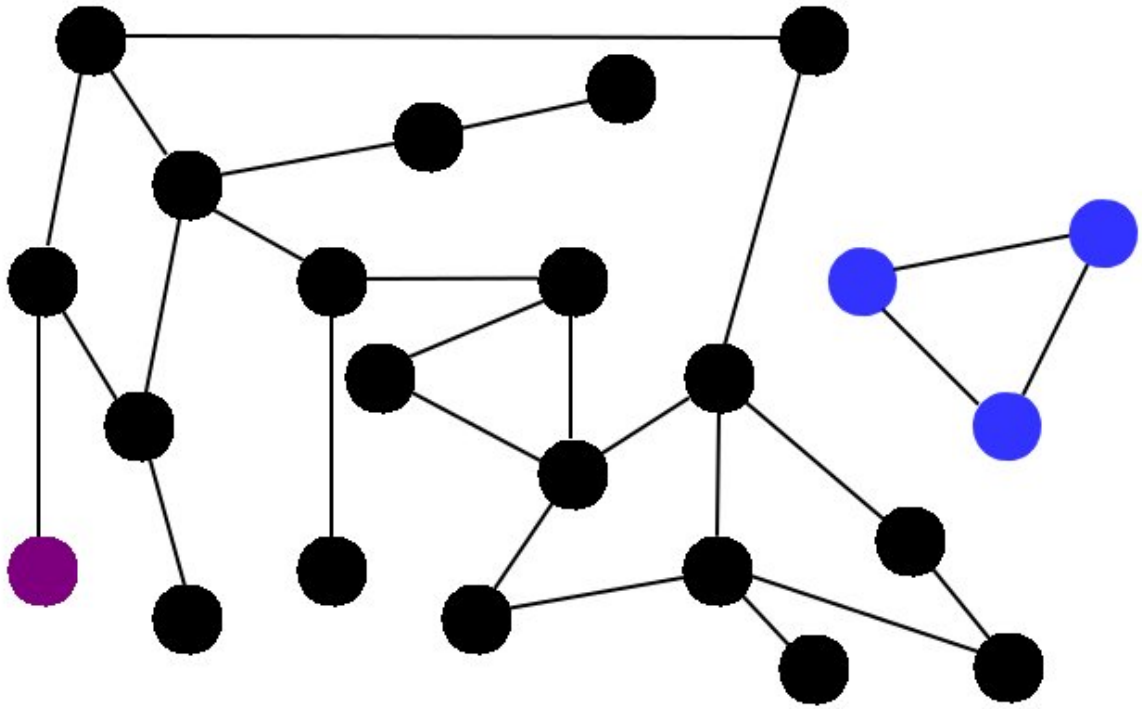
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

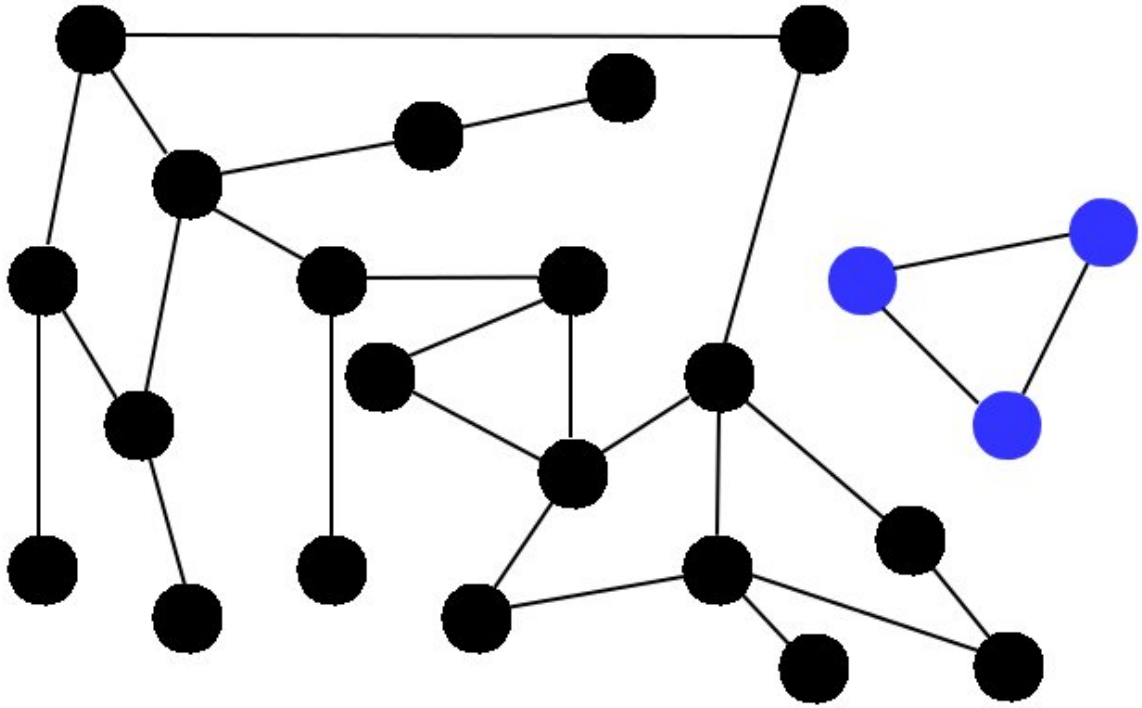
- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

A graph has to be traversed in order to determine its properties

- Breadth-First Search (BFS) is an efficient graph traversal procedure
- BFS starts from a source vertex s
- At each vertex u, all vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Breadth-First Search

BFS (Given graph g and source vertex s)

- Initialise a queue $Q = \langle s \rangle$ of vertices
- While Q is not empty
  - Remove vertex u from Q
  - For each vertex v such that g has an edge $\{u, v\}$
    - If v has not yet been visited
      - Add v to Q
- HALT

- The use of a (FIFO) queue is characteristic of a breadth-first procedure

Applications

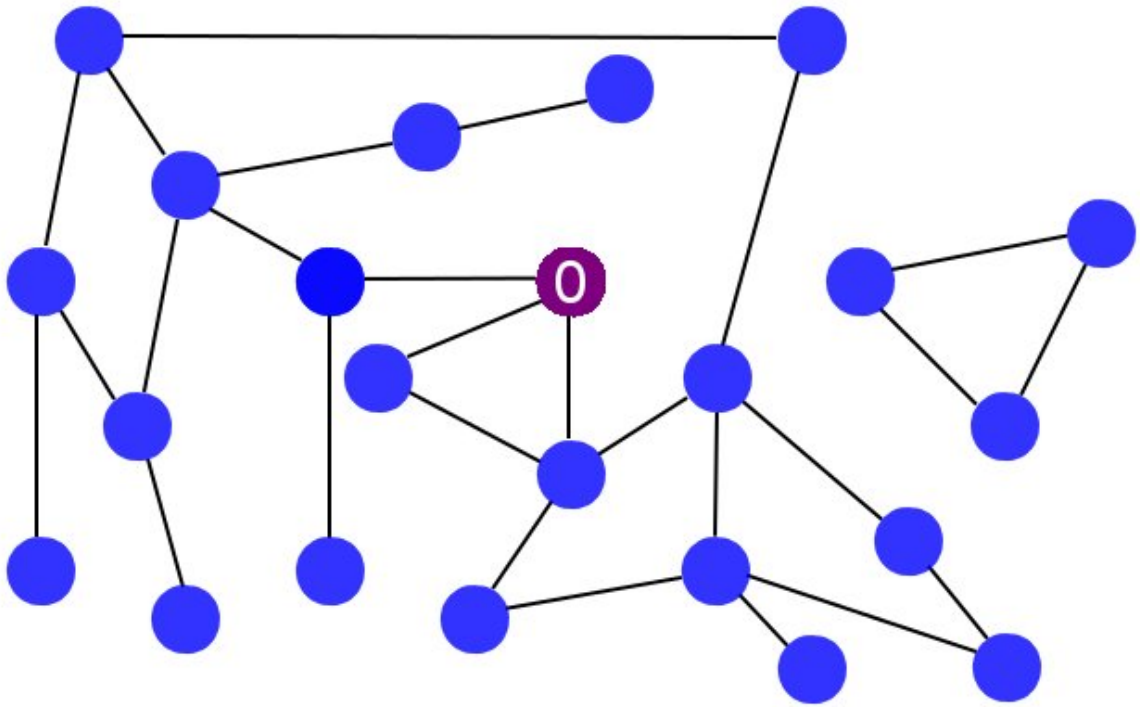
BFS can be used to find cycles, connected components, paths ...

BFS Shortest Paths (Input: graph $G = (V, E)$, source vertex s)

- Initialise a queue $Q = \langle s \rangle$ of vertices
- $dist = \langle d_1, \ldots, d_{|V|} \rangle = \langle \infty, \ldots, \infty \rangle$
- $d_s = 0$
- While Q is not empty
  - Remove vertex u from Q
  - For each vertex v such that g has an edge $\{u, v\}$
    - If v has not yet been visited
      - $d_v = d_u + 1$
      - Add v to Q
- HALT

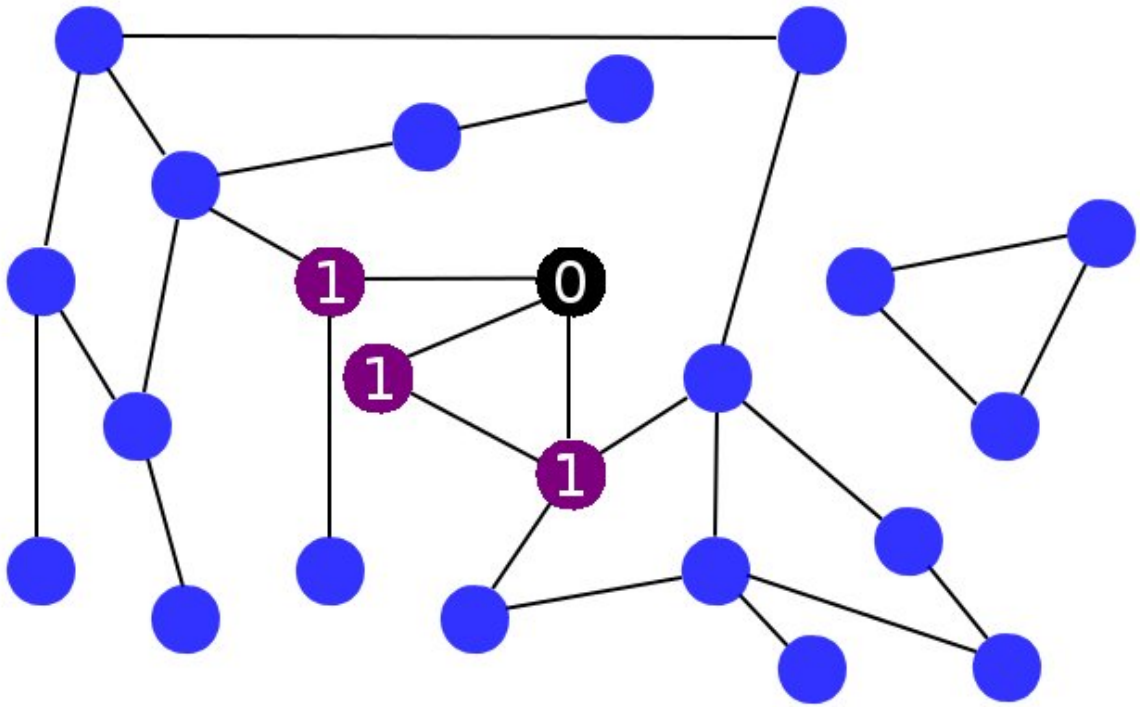
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



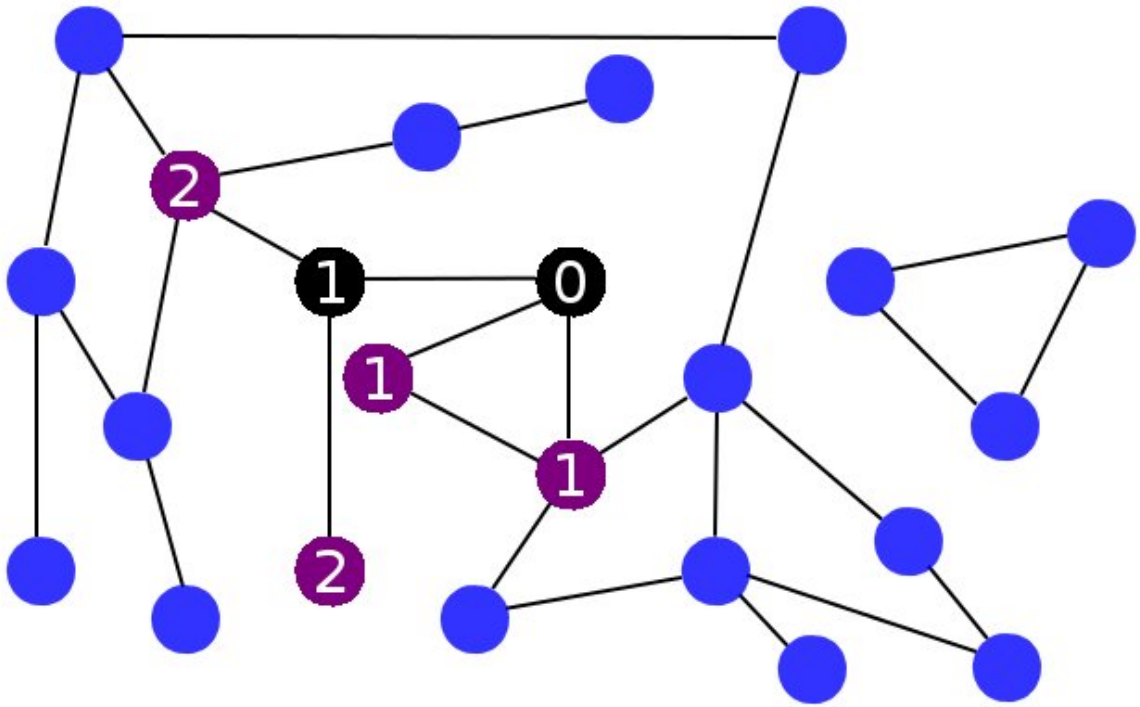
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



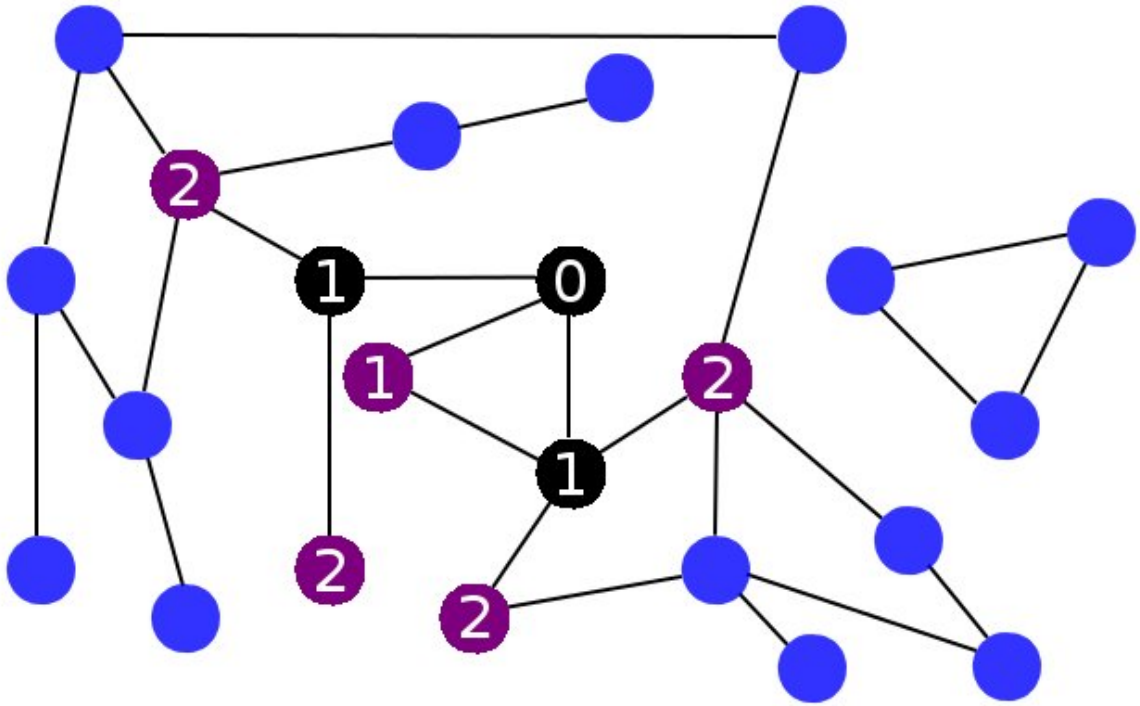
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



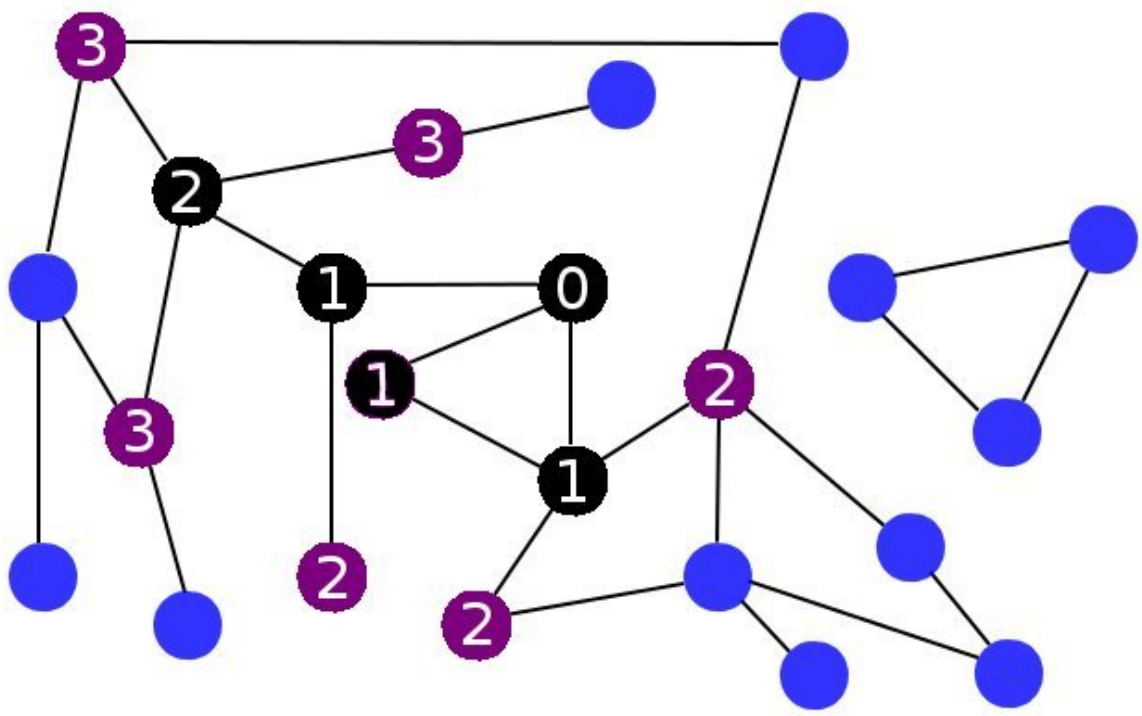
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



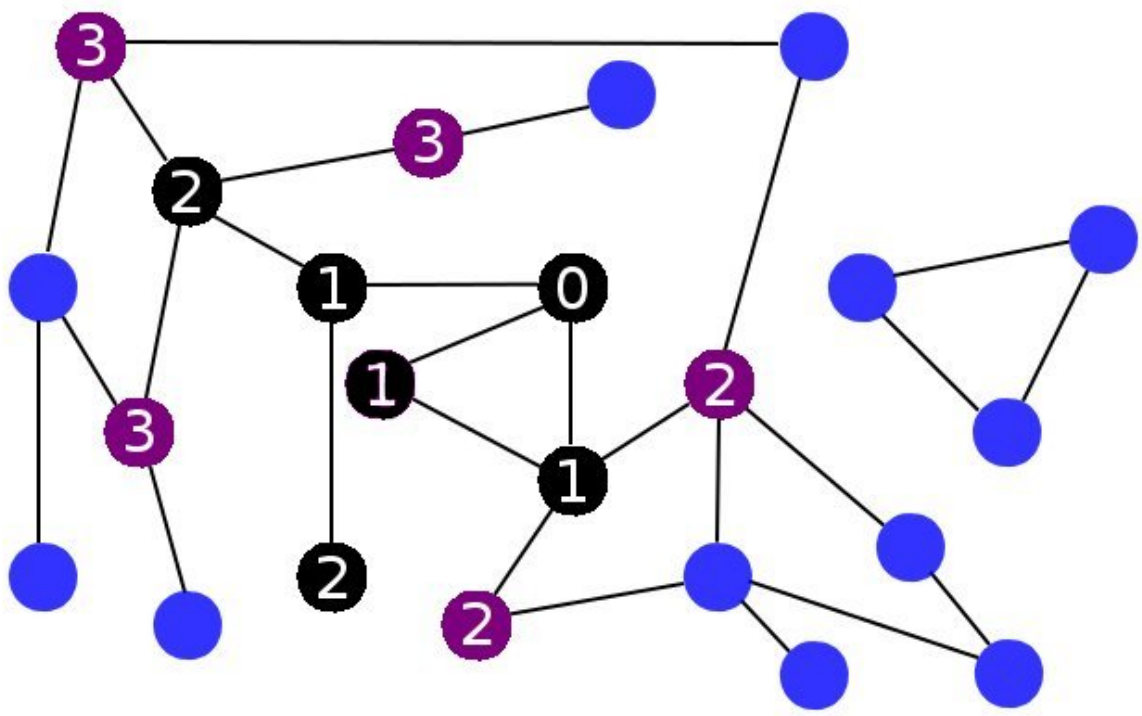
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



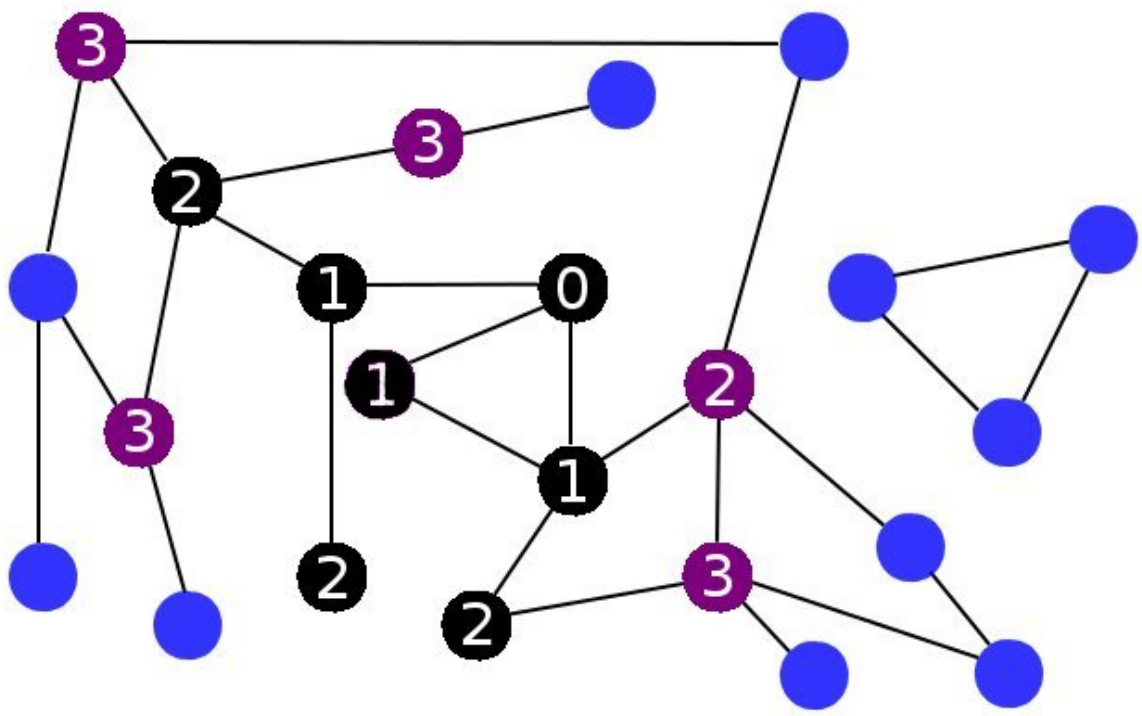
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



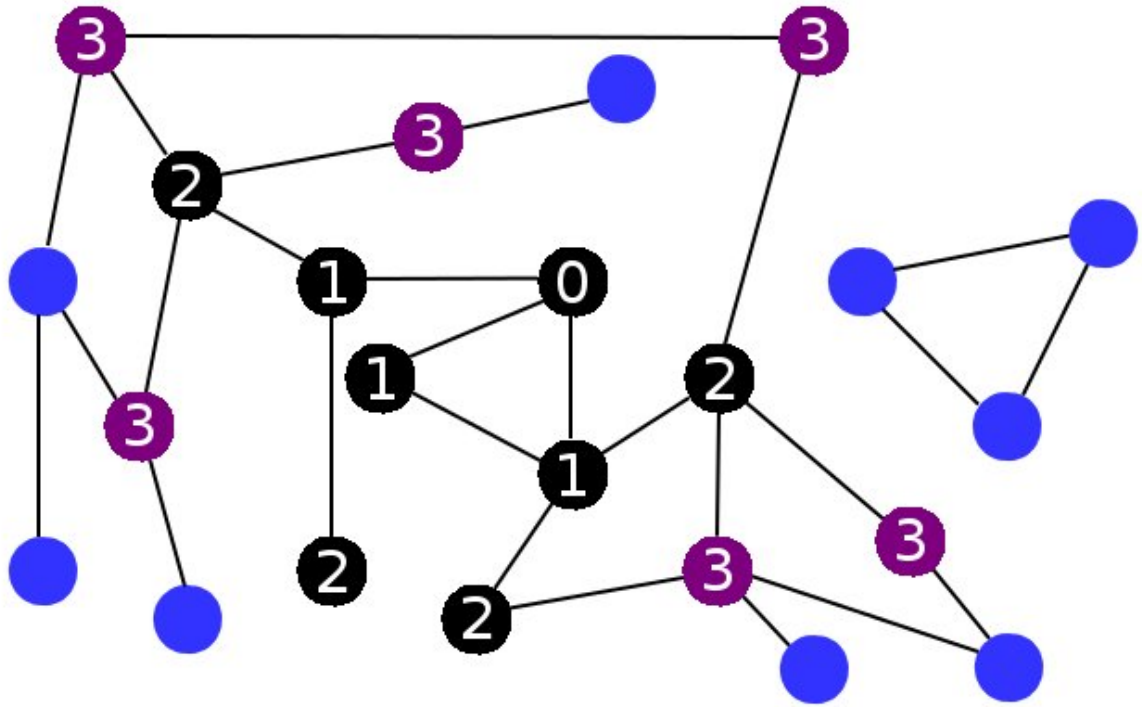
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



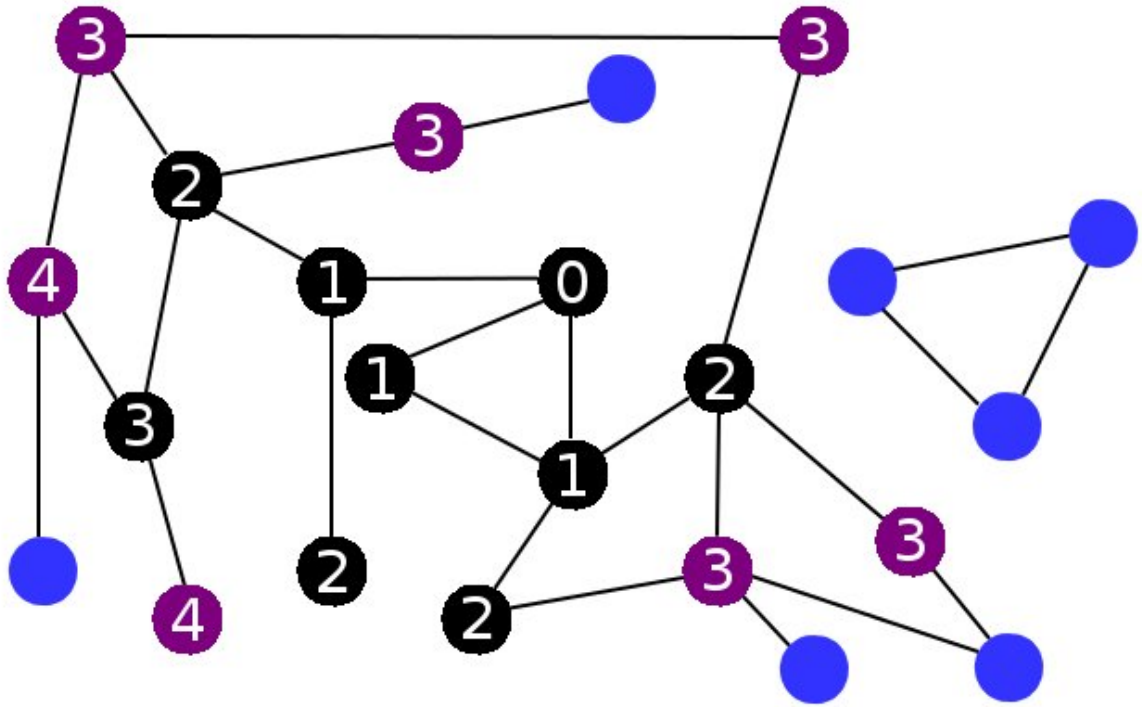
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



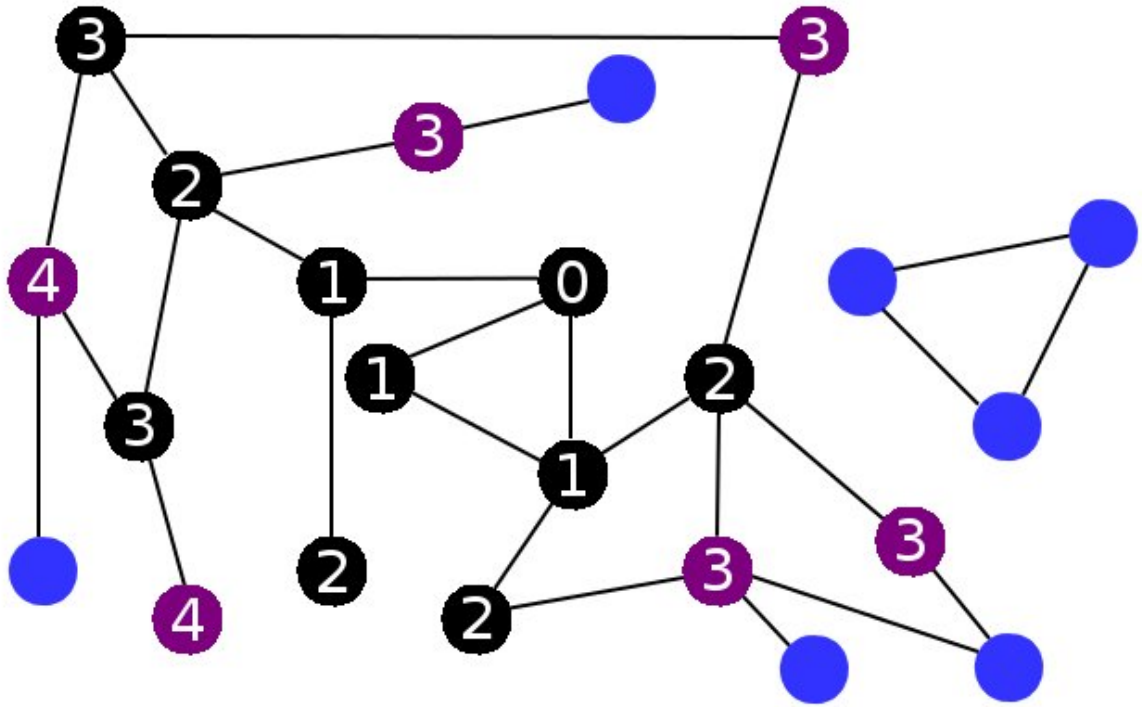
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



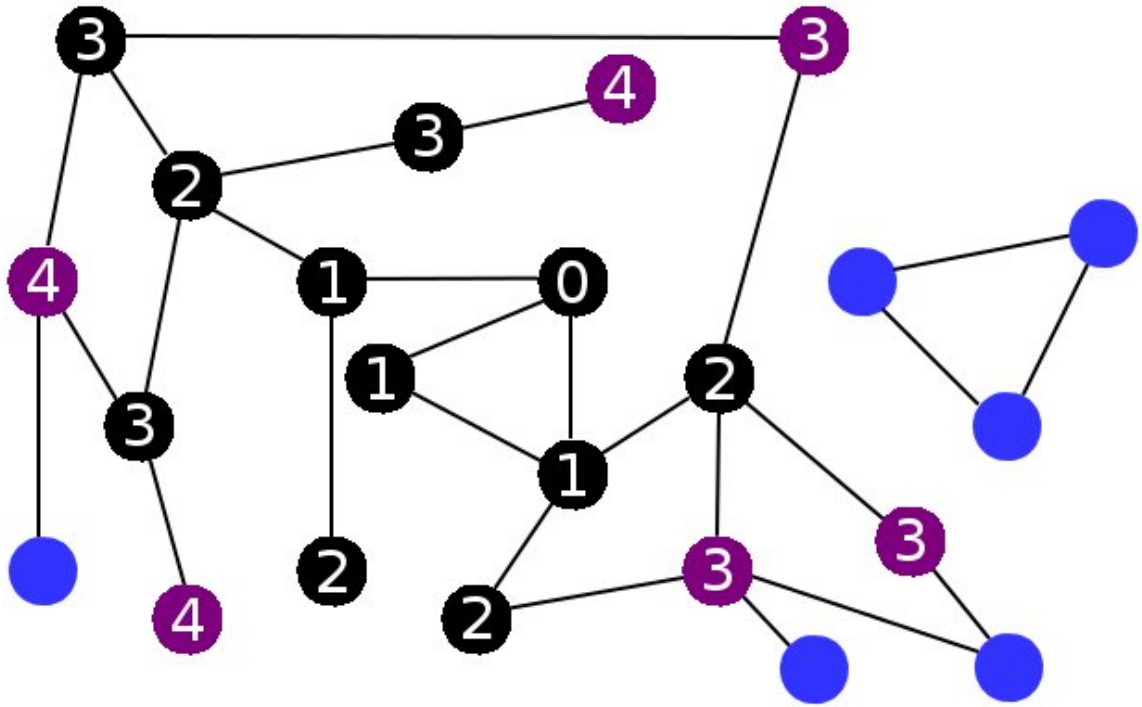
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



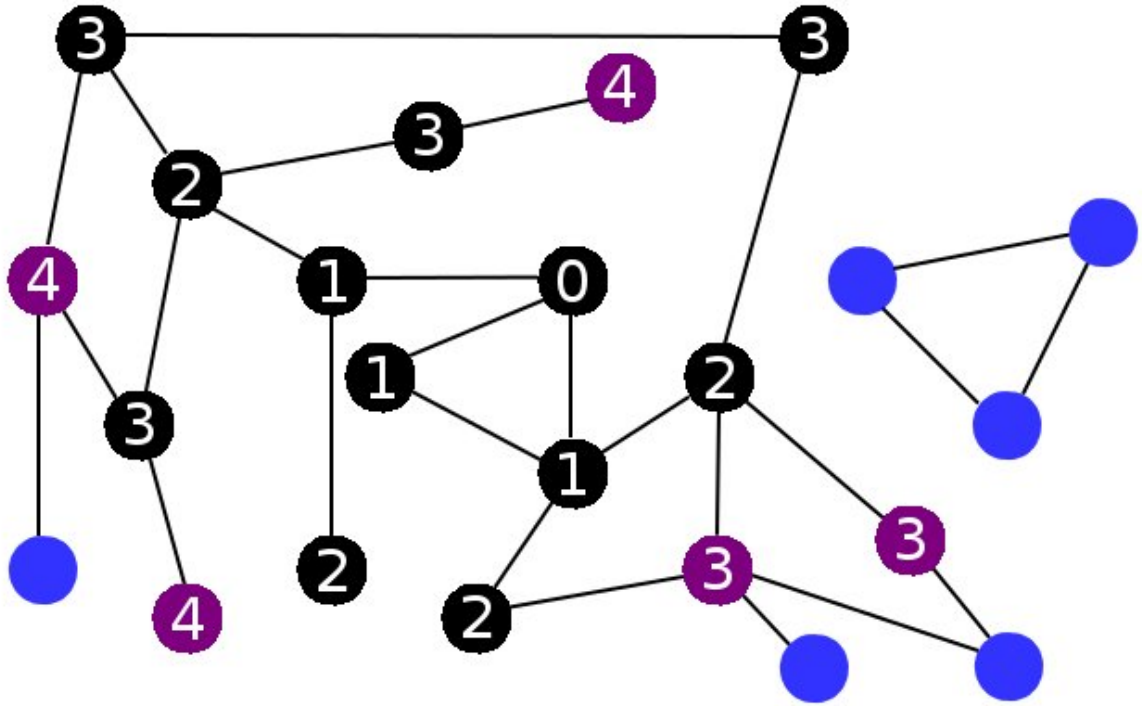
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



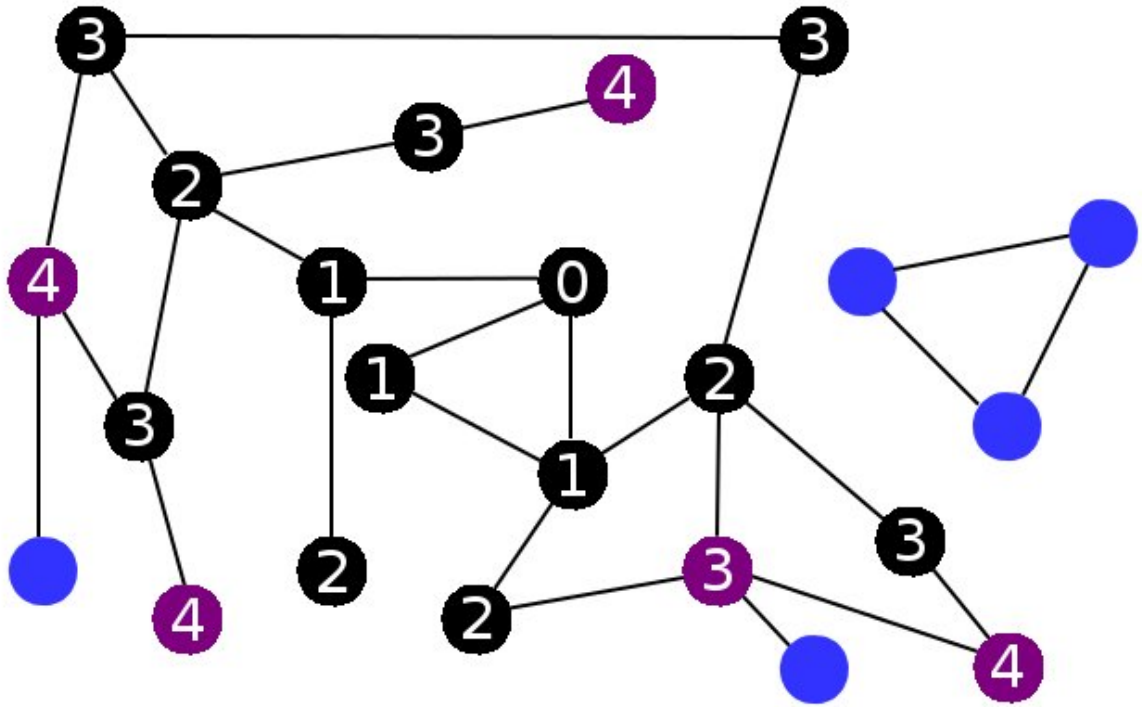
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



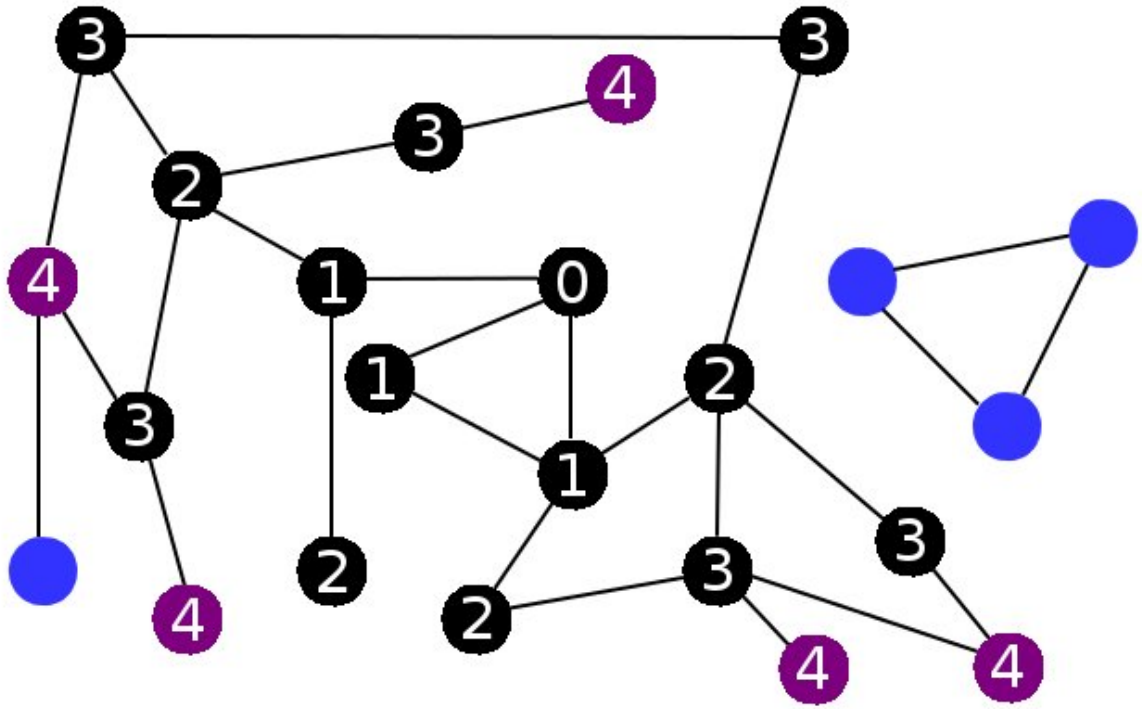
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



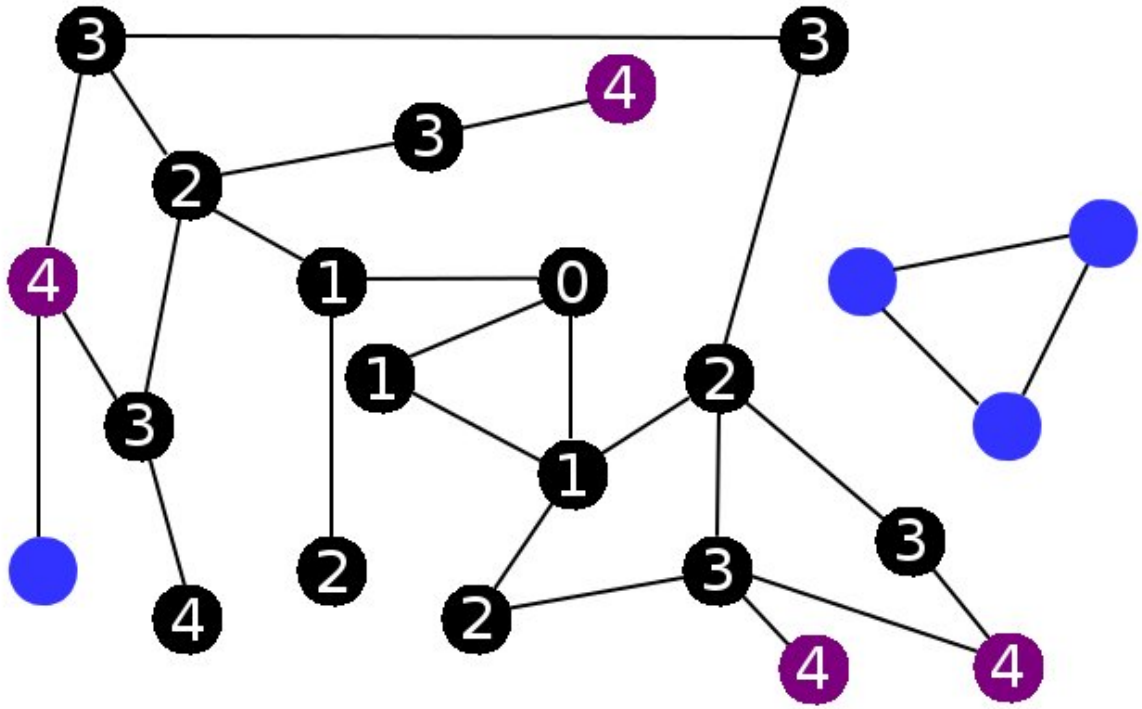
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



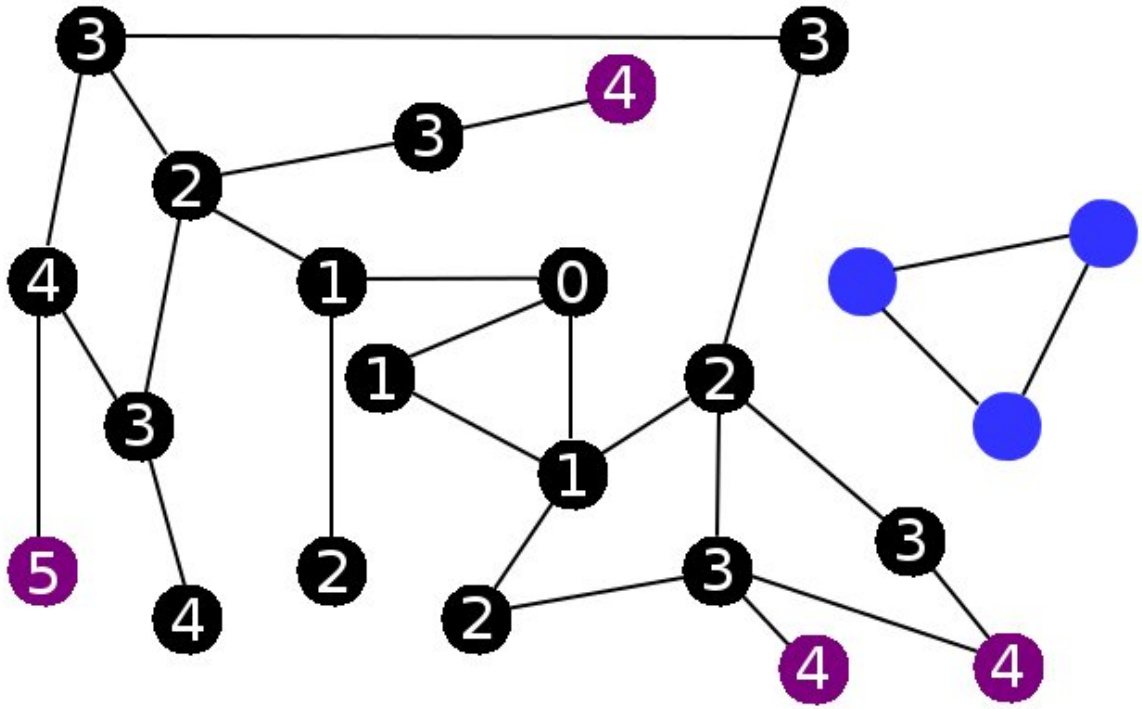
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



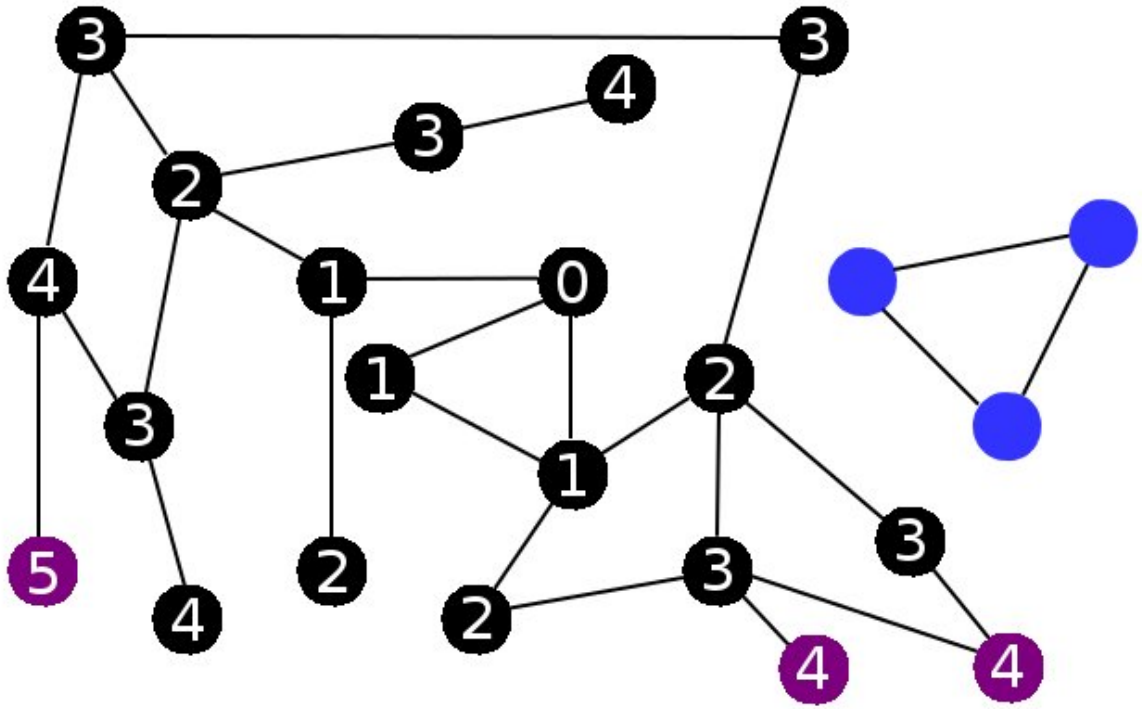
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



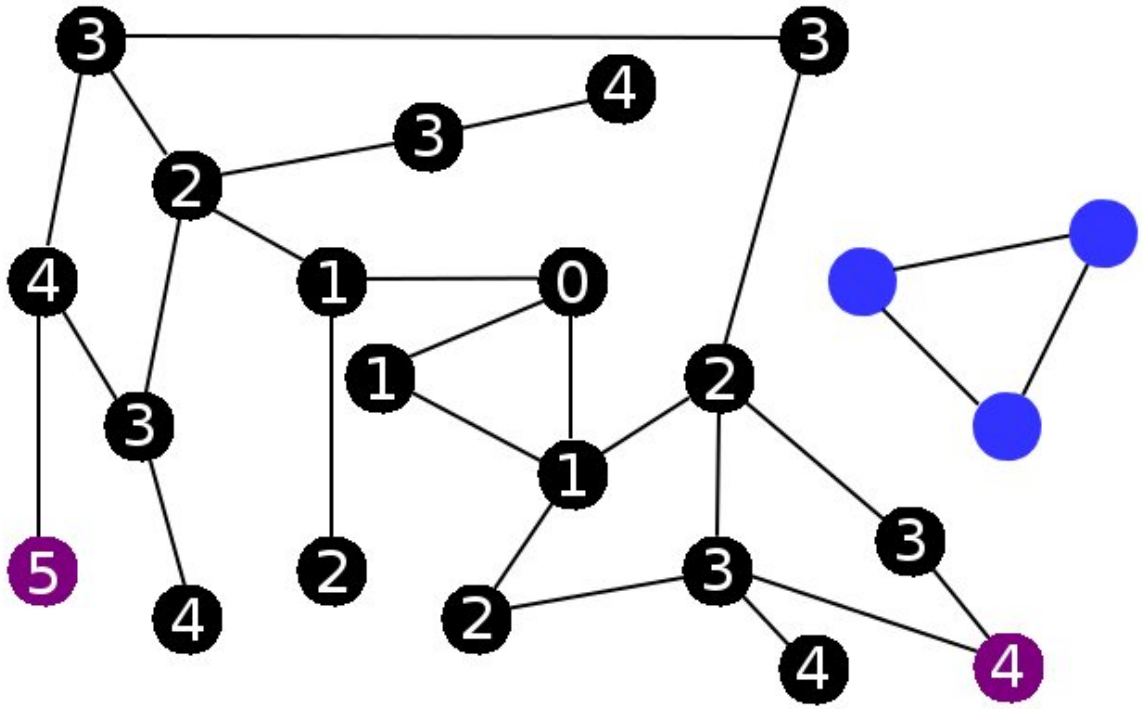
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



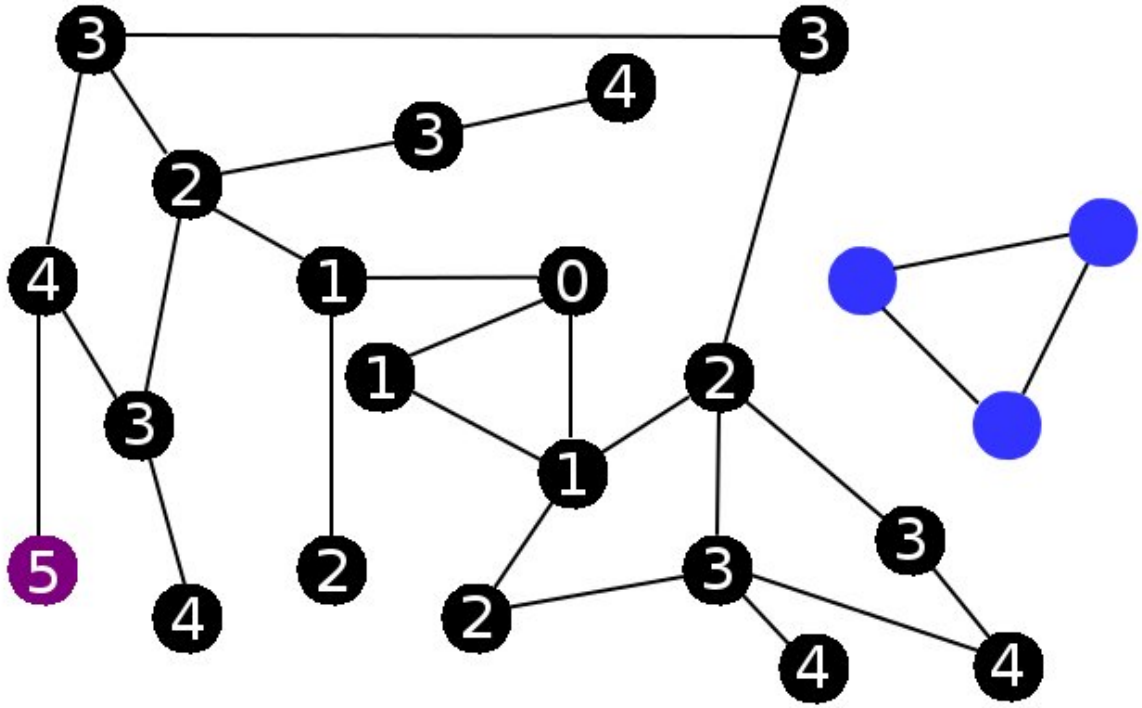
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



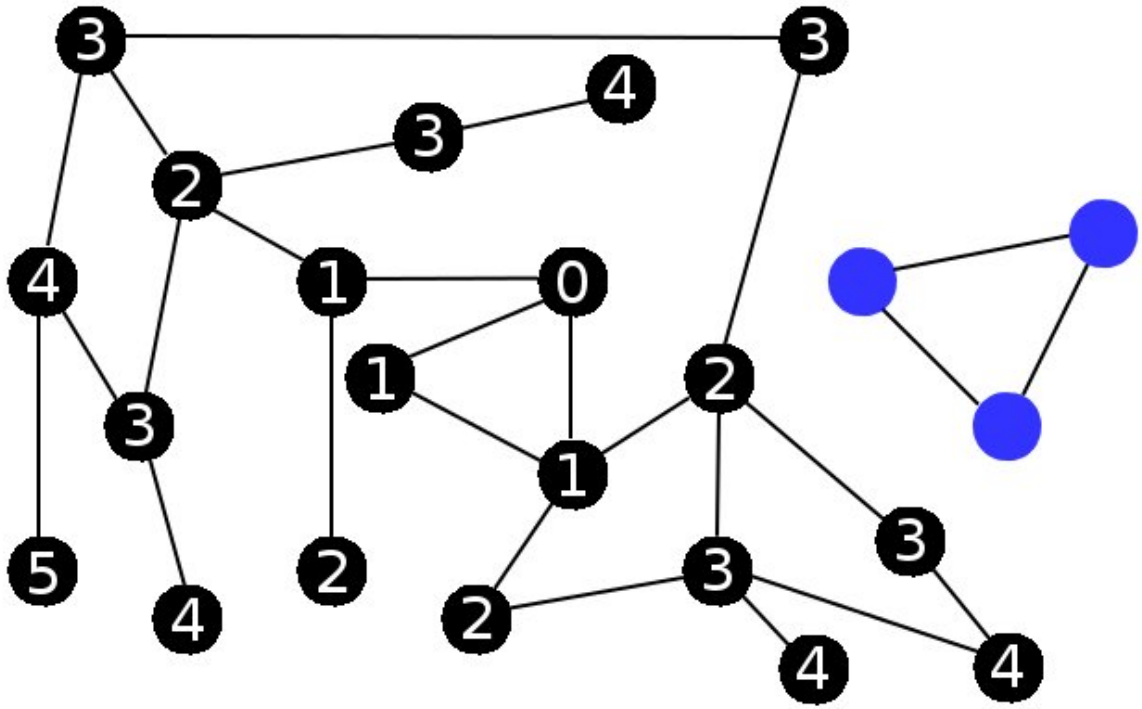
Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



Breadth-First Search

- The distance is recorded when a vertex is (first) visited
- Arrays of size $|V|$ like *dist* are also common in graph search



Implementing a Graph

- This class uses an adjacency list representation
- Vertices are simply indexes $0 \ldots |V| - 1$

```
1  public class Graph {
2
3      List<Integer>[] adj;
4
5      public Graph(int v) {
6          adj = (List<Integer>[]) new List[v];
7          for (int i = 0; i < v; i++) { adj[i] = new List<Integer>(); }
8      }
9
10     public void addEdge(int i, int j) {
11         if (!isVertex(i) || ! isVertex(j)) {
12             throw new IndexOutOfBoundsException();
13         }
14         adj[i].add(j);
15         adj[j].add(i);
16     }
17
18     public Iterable<Integer> adjacent(int v) {
19         return adj[v];
20     }
21
22     private boolean isVertex(int v) {
23         return 0 <= v && v < adj.length;
24     }
25 }
```

Implementing BFS

- This class searches the given Graph from vertex s
- The shortest distance to vertex `v` is saved in `distance[v]`
- The previous node on the path to `v` is saved in `parent[v]`

```
1   public class BFS {
2
3       private Graph graph;
4       private int[] distance;
5       private int[] parent;
6
7       public BFS(Graph g, int s) {
8           graph     = g;
9           int v     = g.vertices();
10          distance  = new int[v];
11          parent    = new int[v];
12          Arrays.fill(distance, -1);
13          Arrays.fill(parent, -1);
14          bfs(s);
15      }
16
17      ...
```

Implementing BFS

- The search keeps track of visited vertices using a `boolean[]`
- When `u` is removed from the queue its adjacency list is acquired

```
1       ...
2
3       private void bfs(int s) {
4           boolean[] visited = new boolean[graph.vertices()];
5
6           Queue<Integer> q  = new Queue<Integer>();
7           q.add(s);
8           distance[s] = 0;
9           visited[s]  = true;
10
11          while(!q.isEmpty()) {
12              int u = q.remove();
13              for (int v : graph.adjacent(u)) {
14                  if (!visited[v]) {
15                      visited[v]  = true;
16                      parent[v]   = u;
17                      distance[v] = distance[u] + 1;
18                      q.add(v);
19                  }
20              }
21          }
22      }
23
24      ...
```

Implementing BFS

- The class also provides methods that use the search results

```
1       ...
2
3       public int distance(int v) {
4           return distance[v];
5       }
6
7       public int[] shortestPath(int v) {
8           if (parent[v] == -1 && distance[v] == -1)  { return null; }
9
10          Stack<Integer> vs = new Stack<Integer>();
11          do {
12              vs.push(v);
13              v = parent[v];
14          } while (v != -1);
15
16          int numVs  = vs.size();
17          int[] path = new int[numVs];
18          for (int i = 0; i < numVs; i++) {
19              path[i] = vs.pop();
20          }
21          return path;
22      }
23
24      public boolean isReachable(int v) {
25          return distance[v] != -1;
26      }
27  }
```

BFS Performance

For a graph with V vertices and E edges BFS runs in $O(V + E)$ time

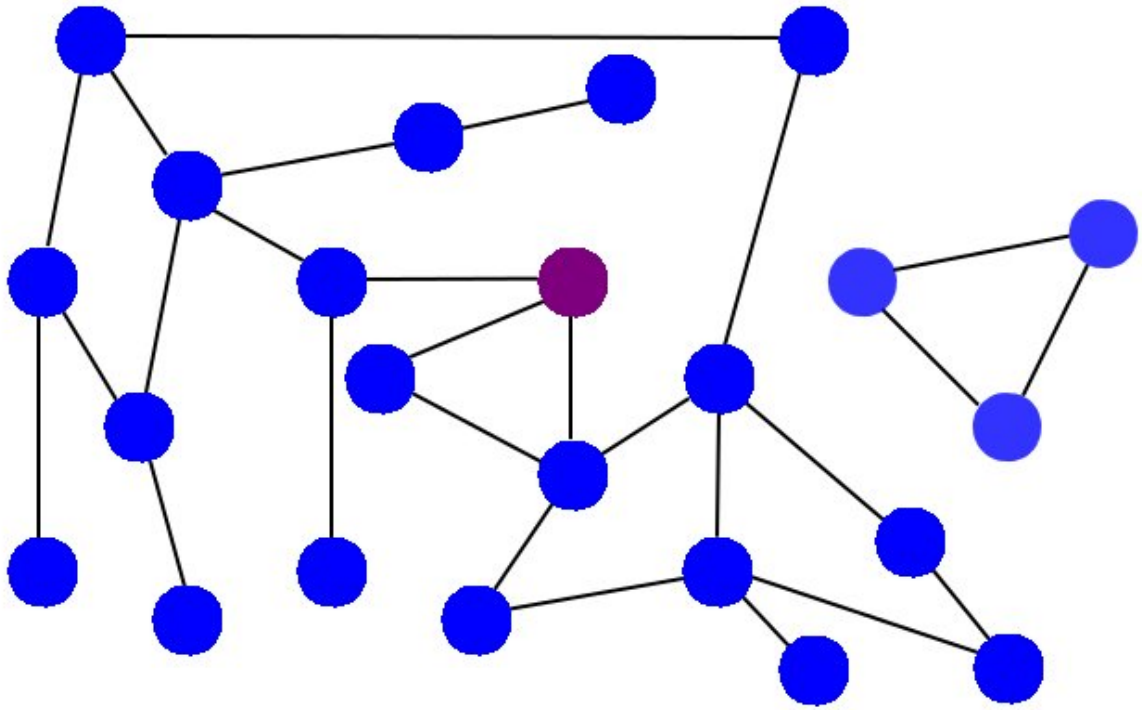
- Assume the graph is connected (worst case)
- Each vertex is added and removed from the queue exactly once
- Each adjacency list is used exactly once
- Each edge contributes exactly two vertices to the adjacency lists

Part II

Depth-First Search

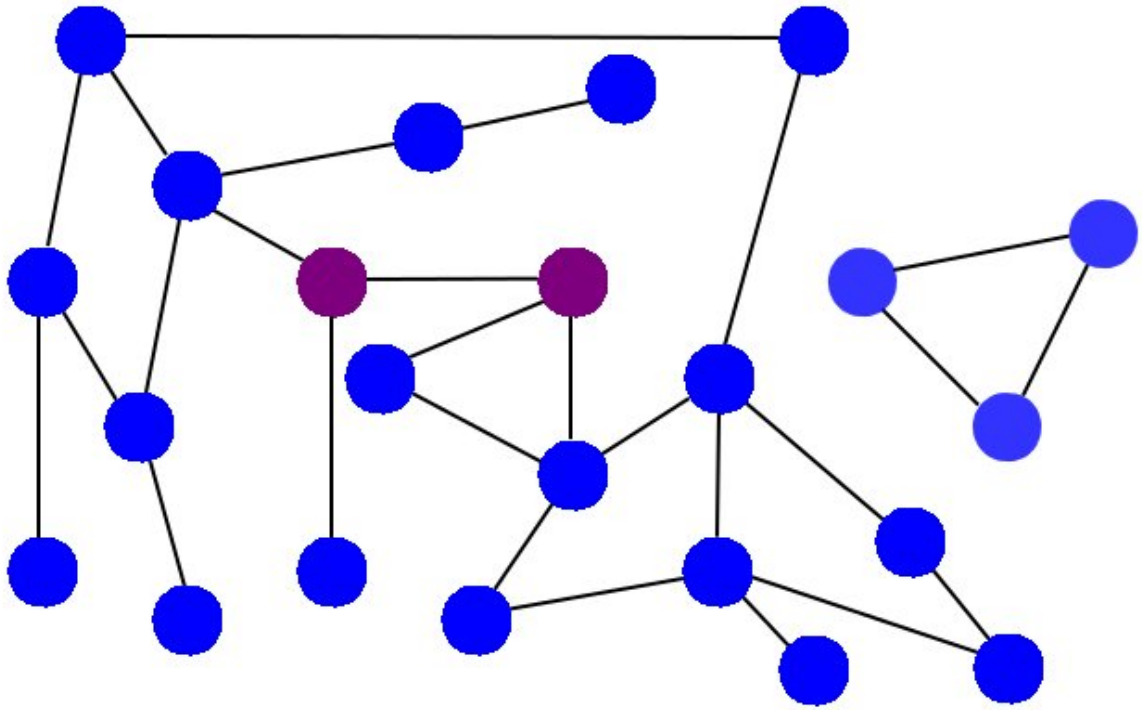
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



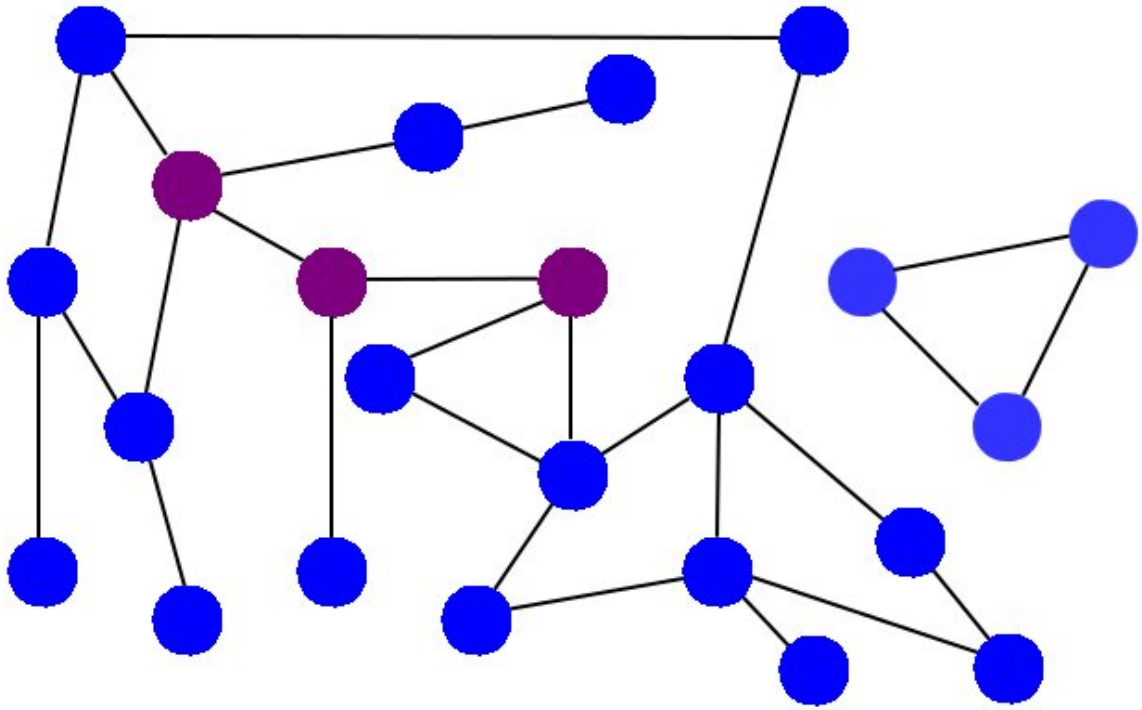
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



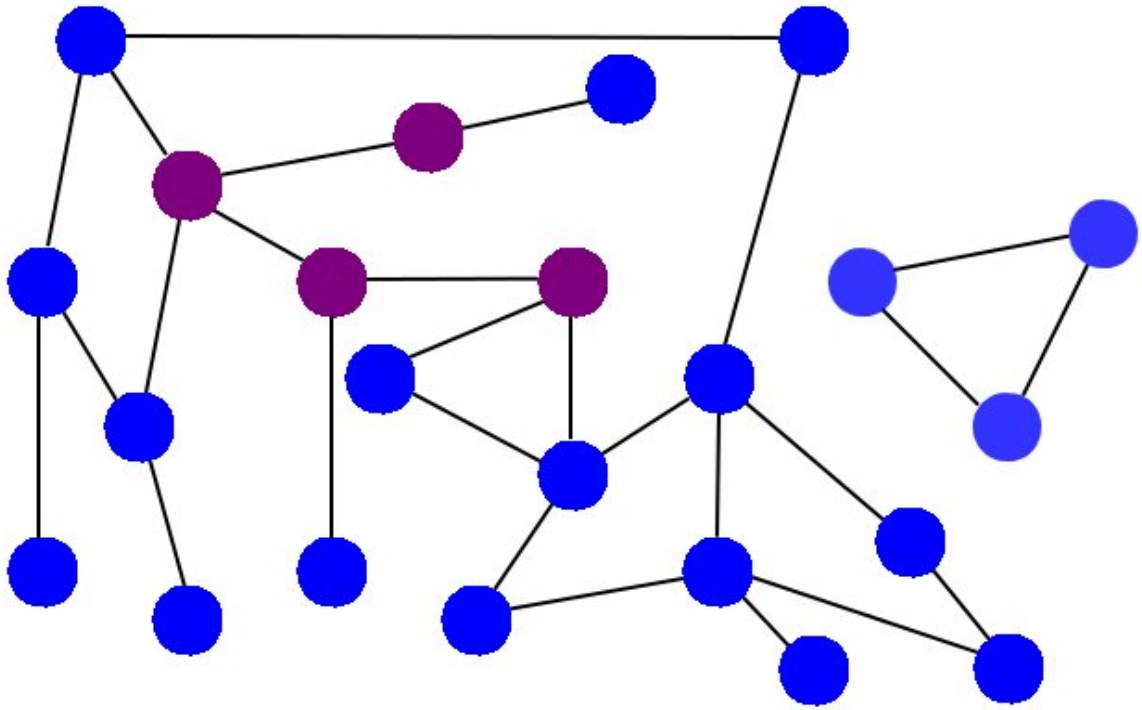
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



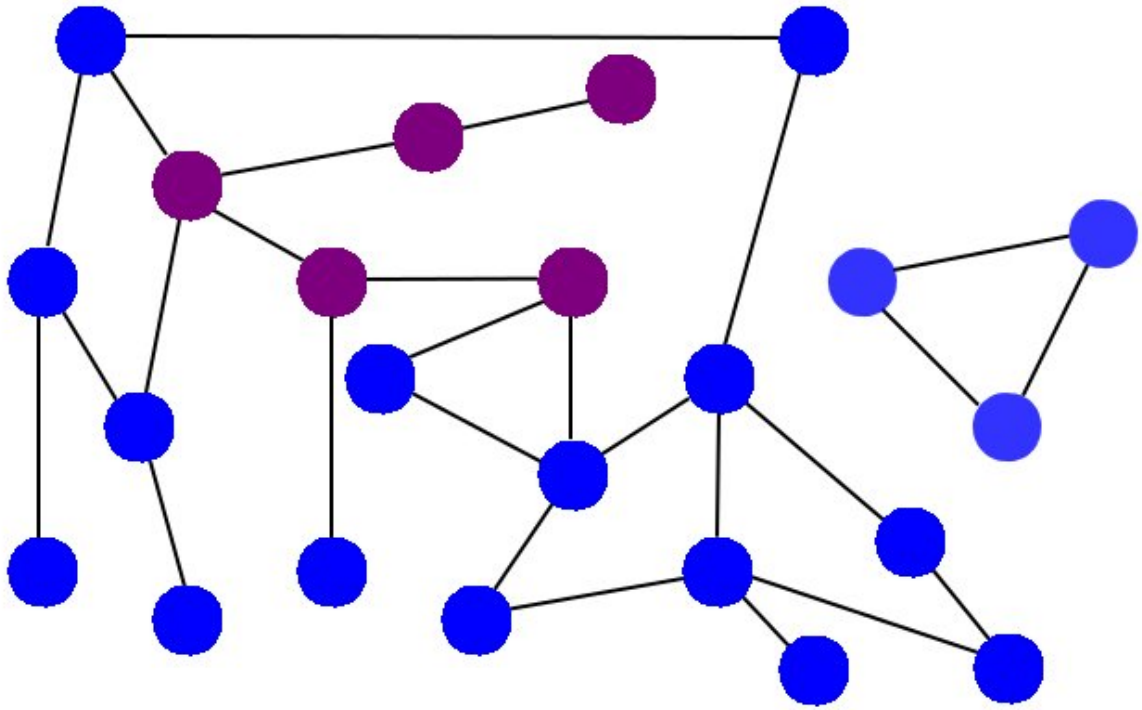
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



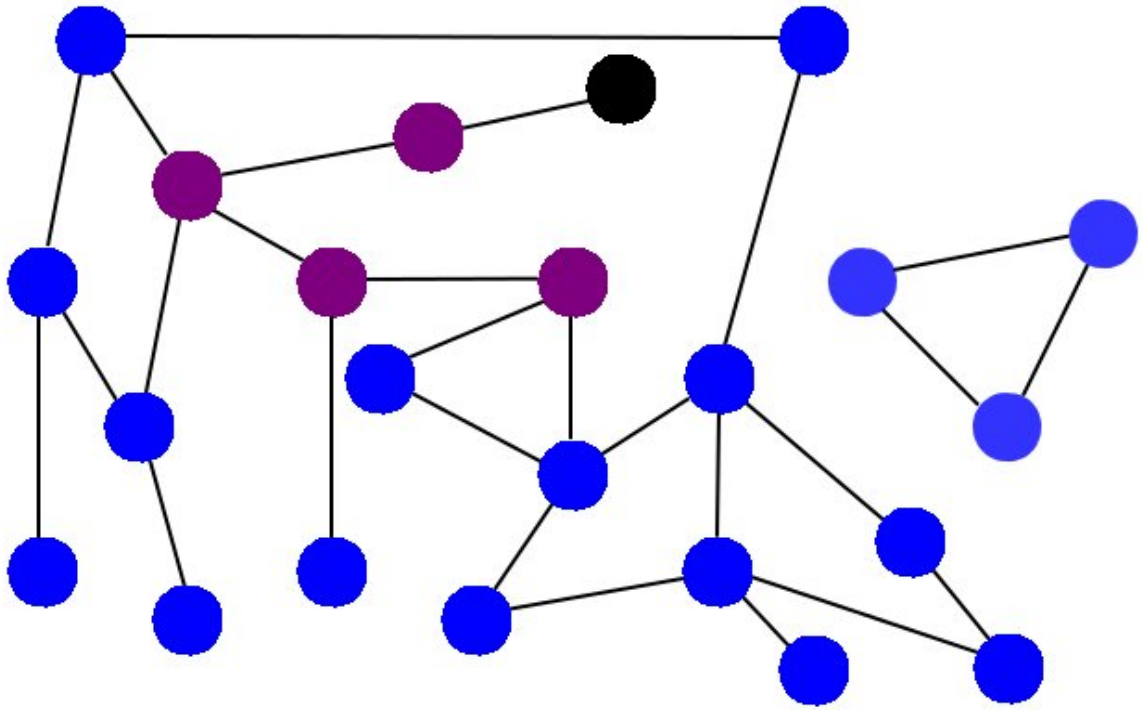
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



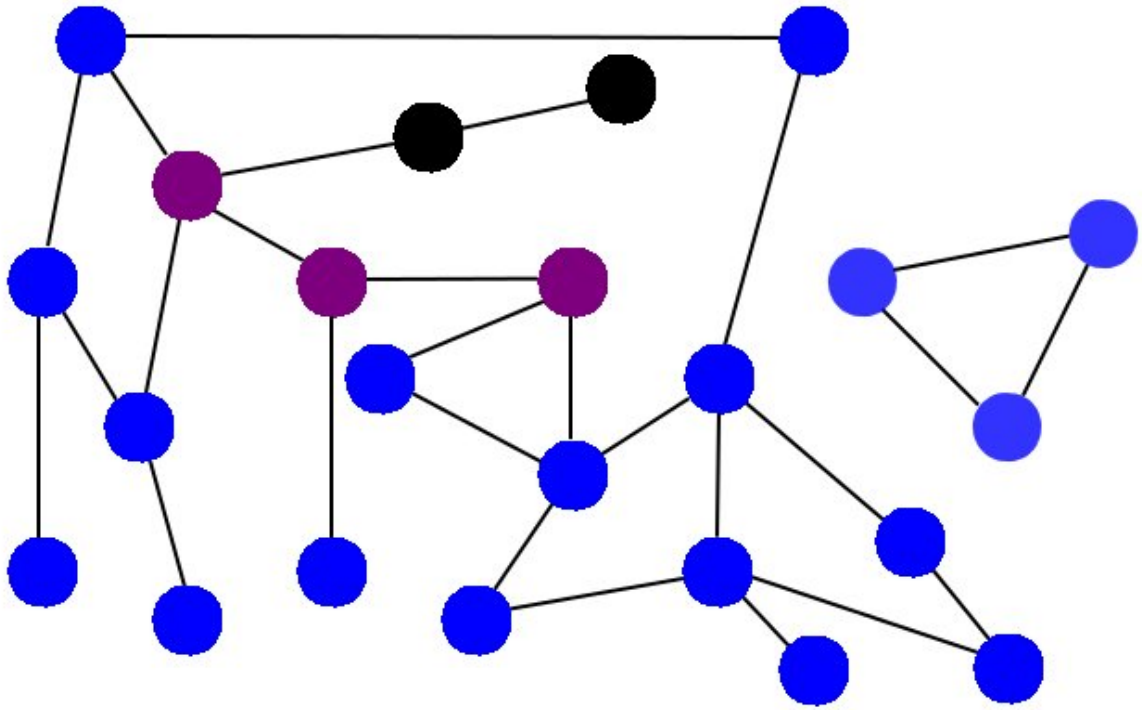
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



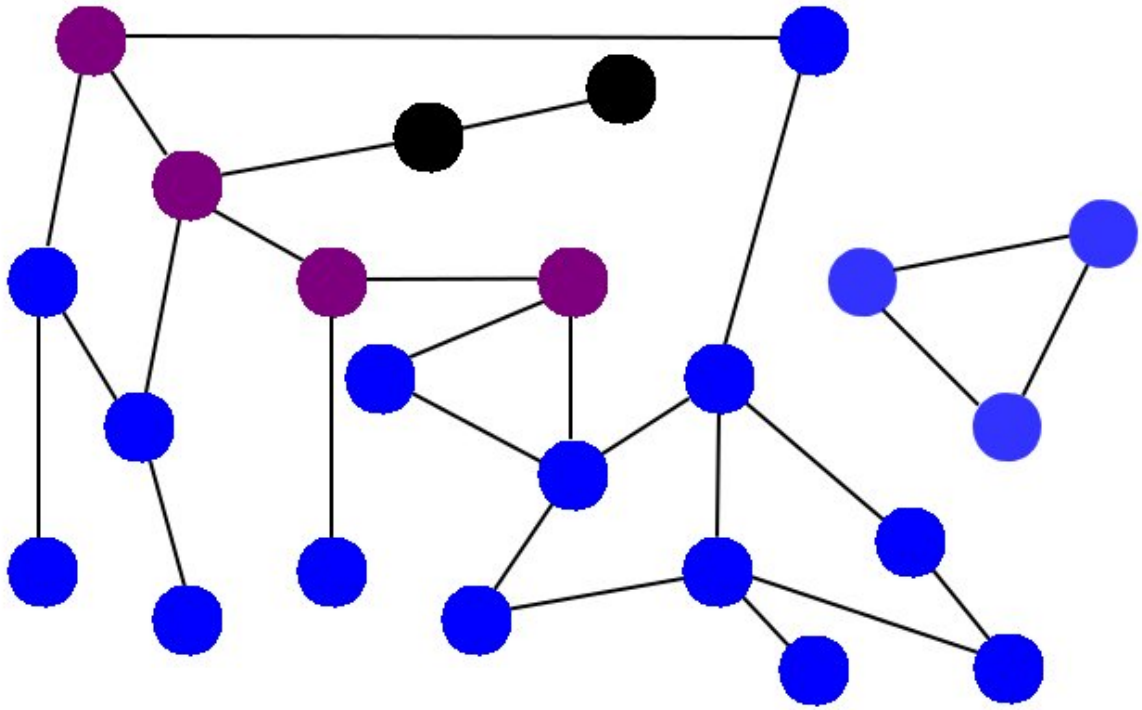
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



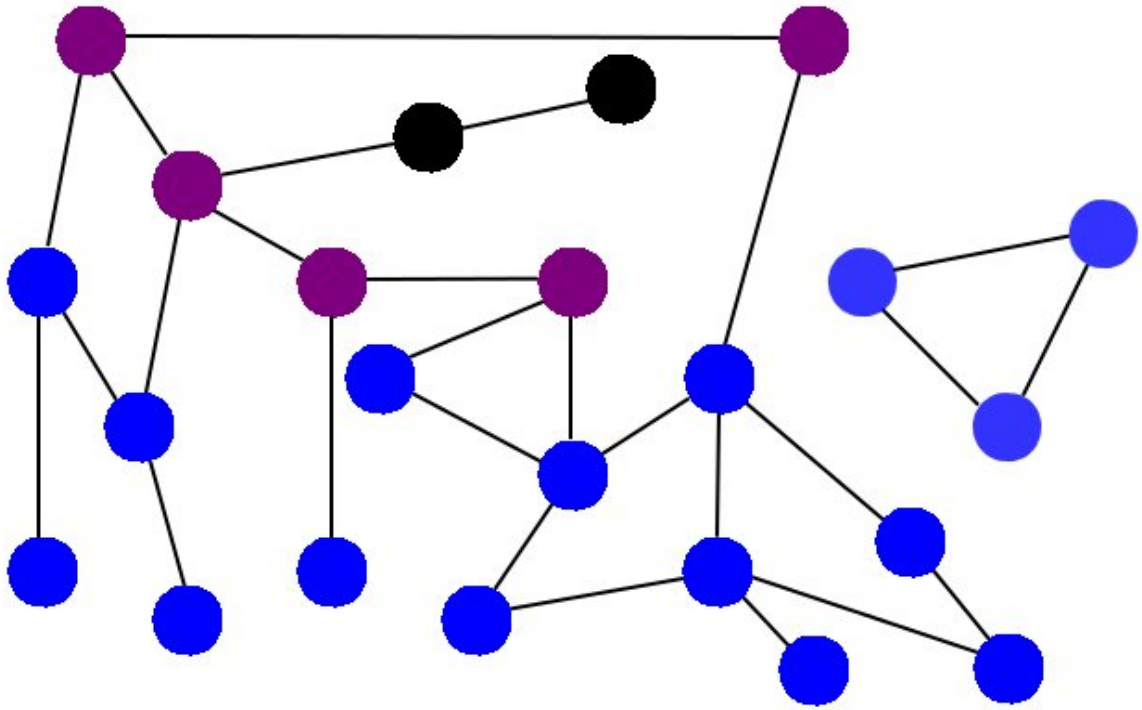
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



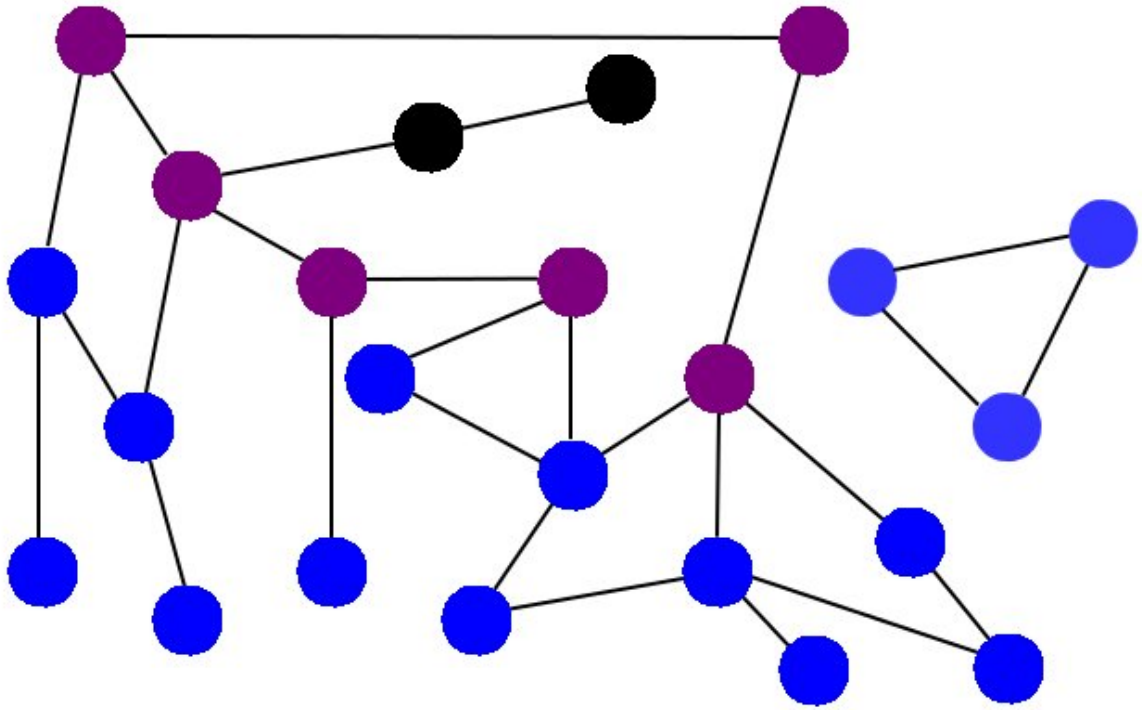
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



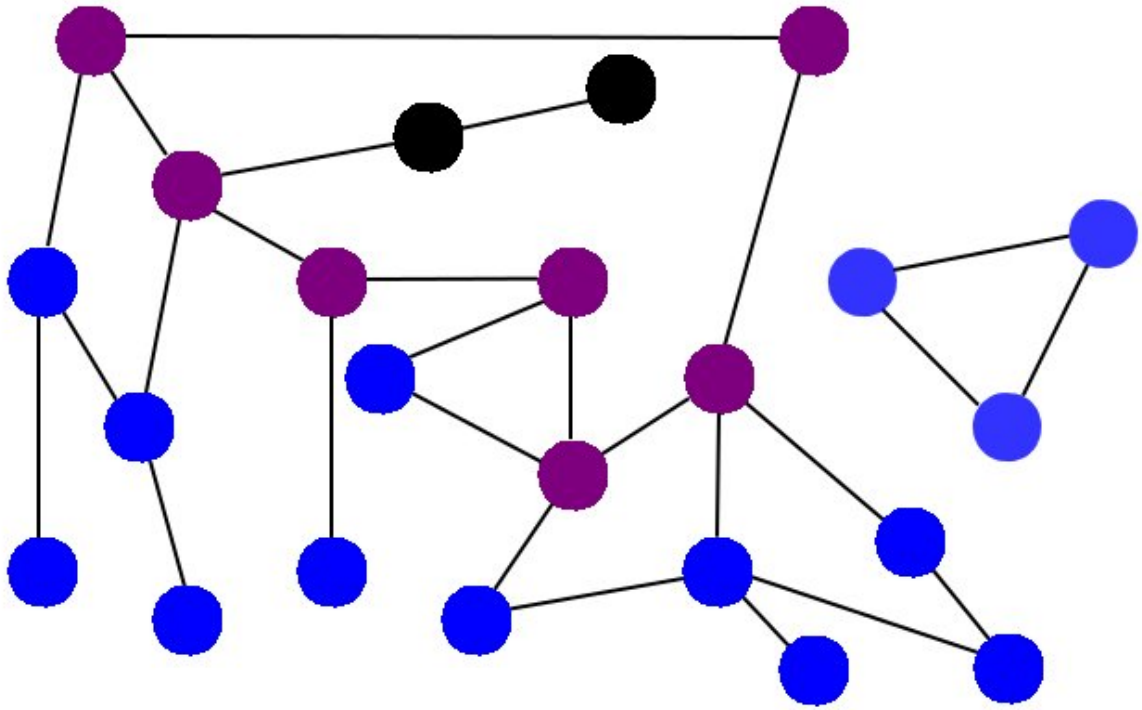
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



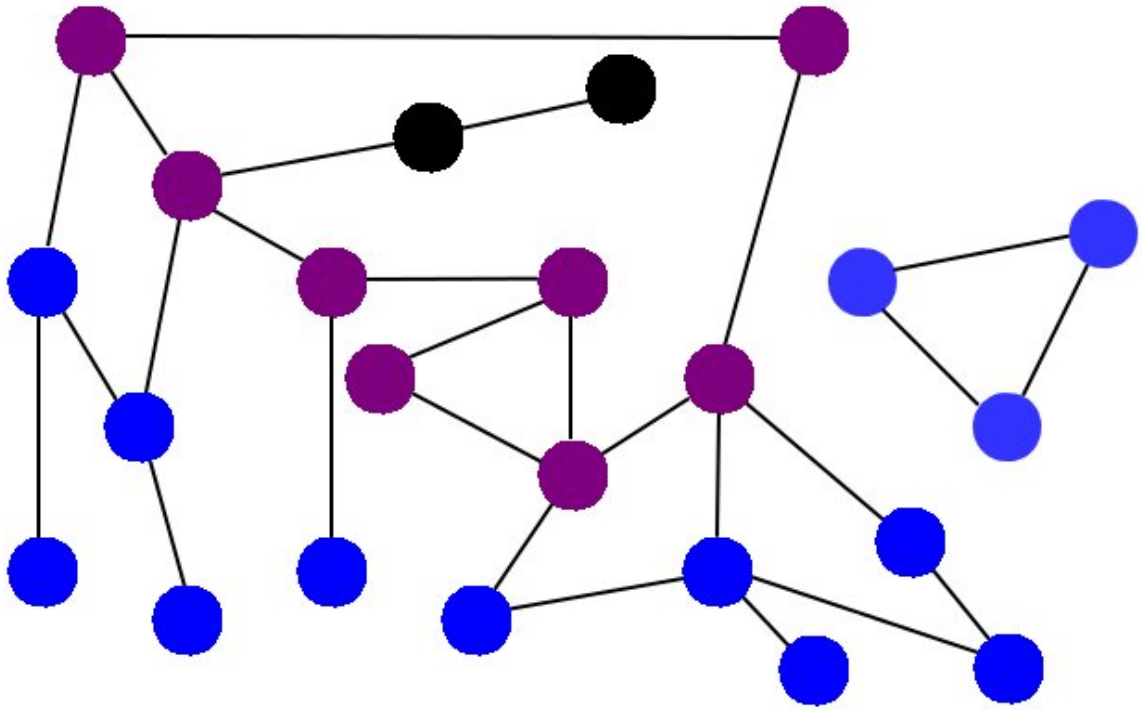
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



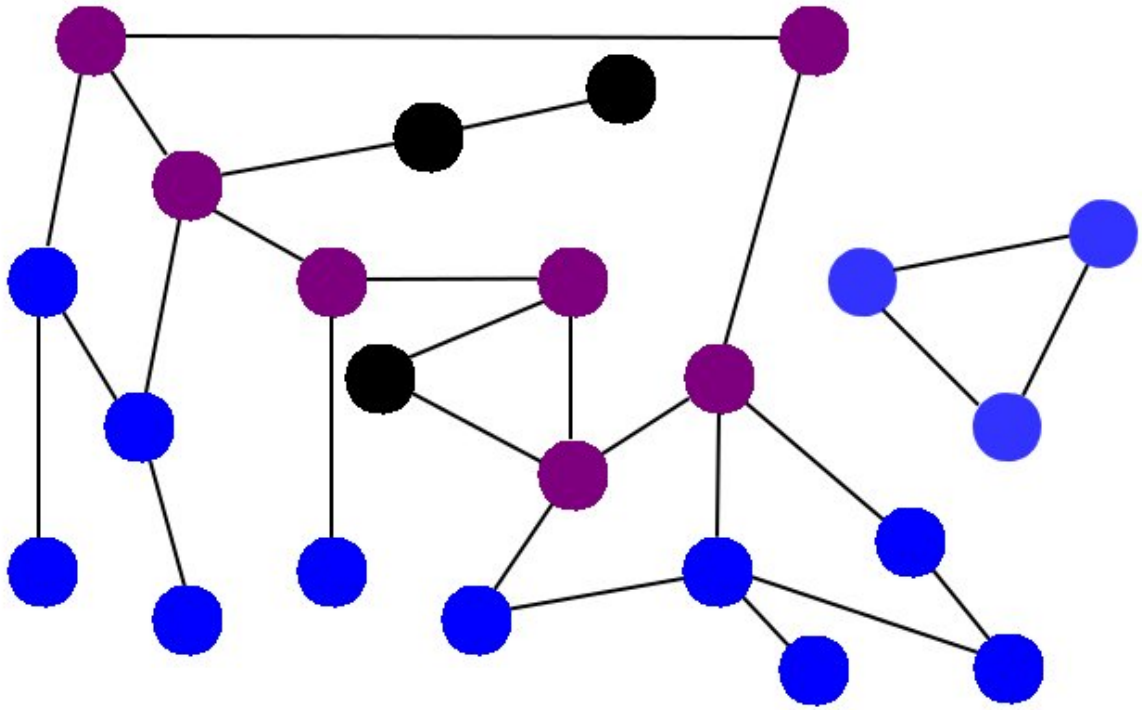
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



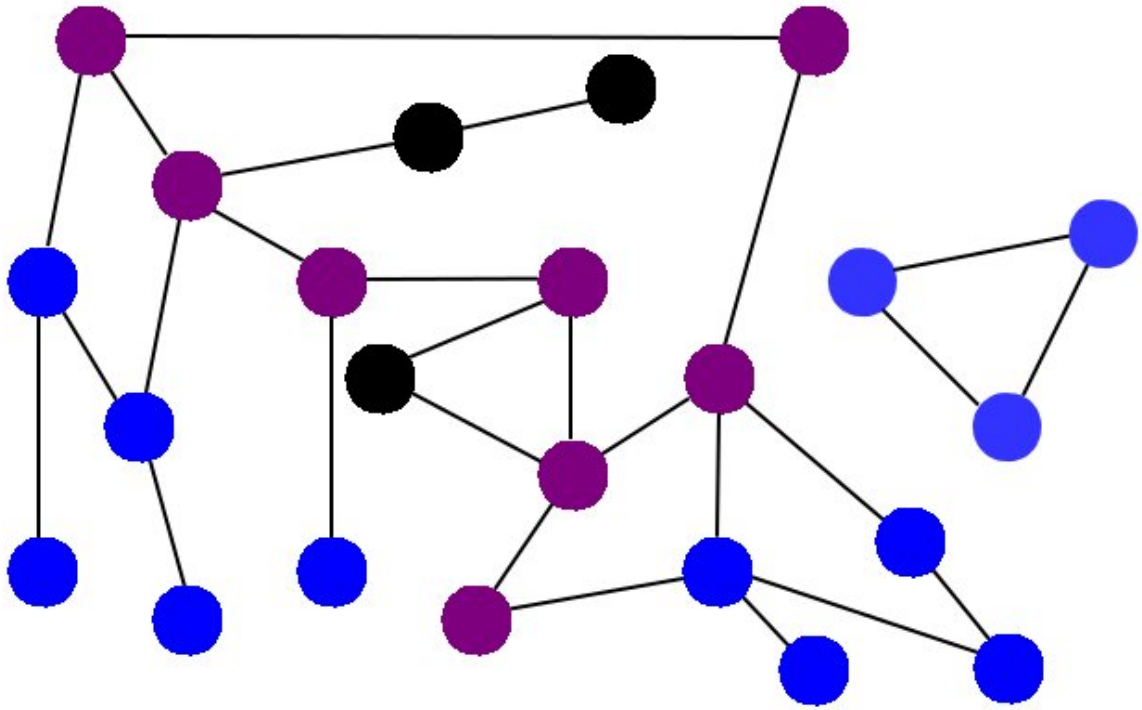
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



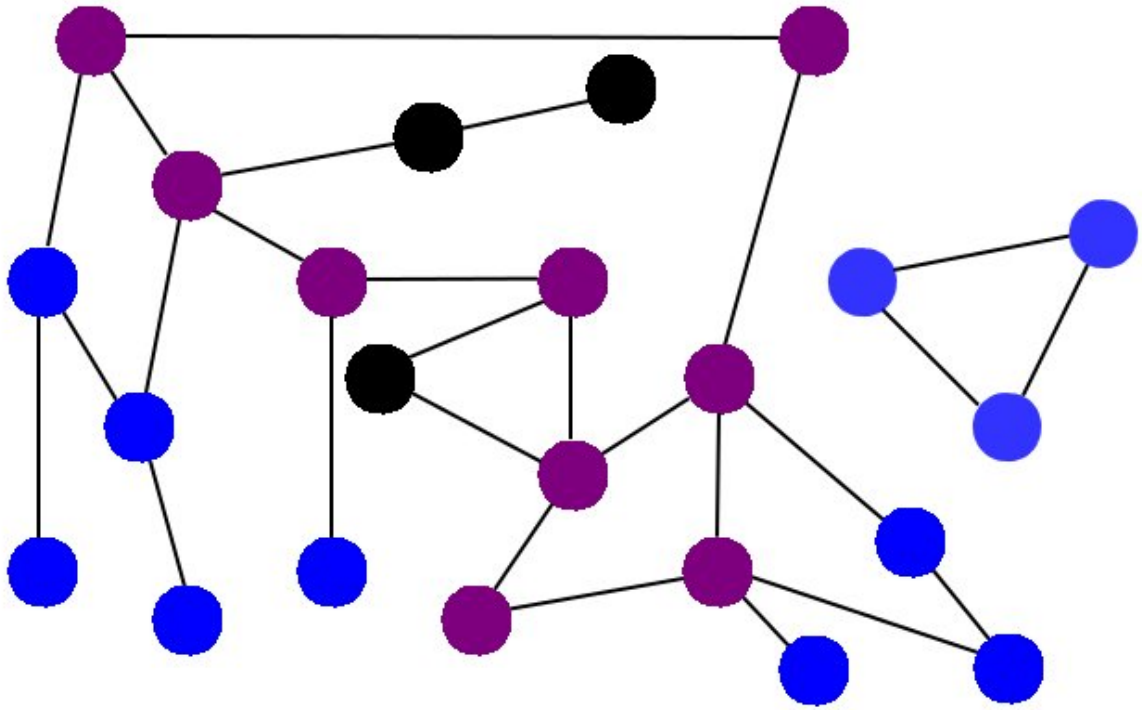
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



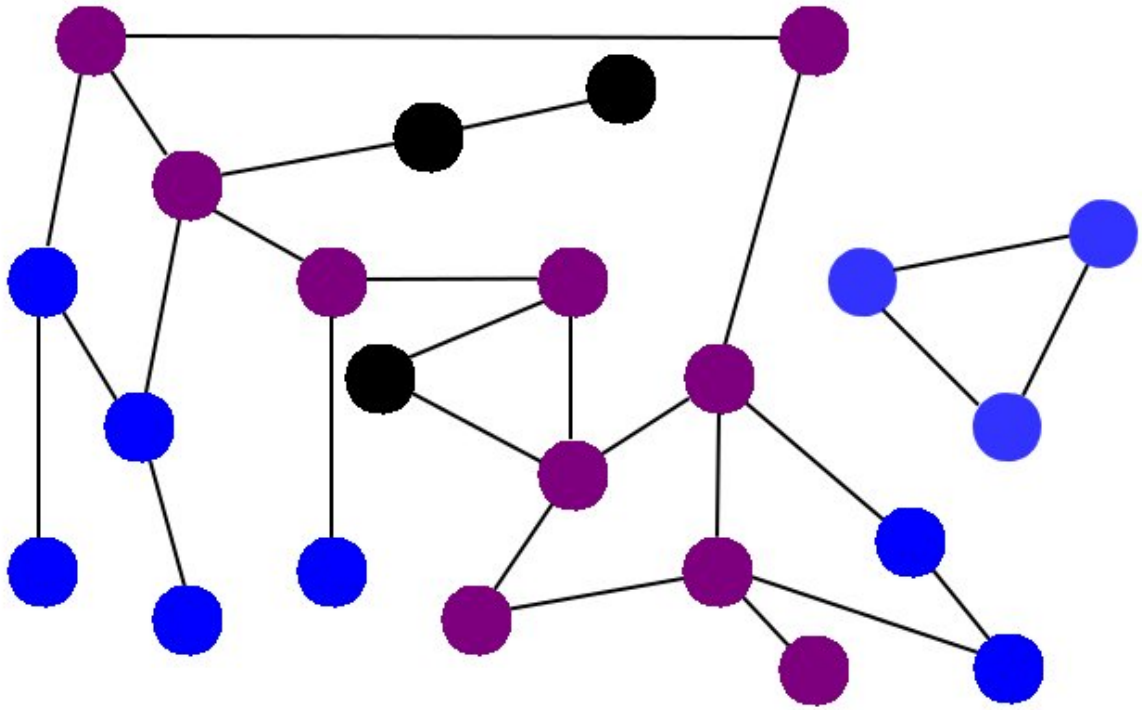
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



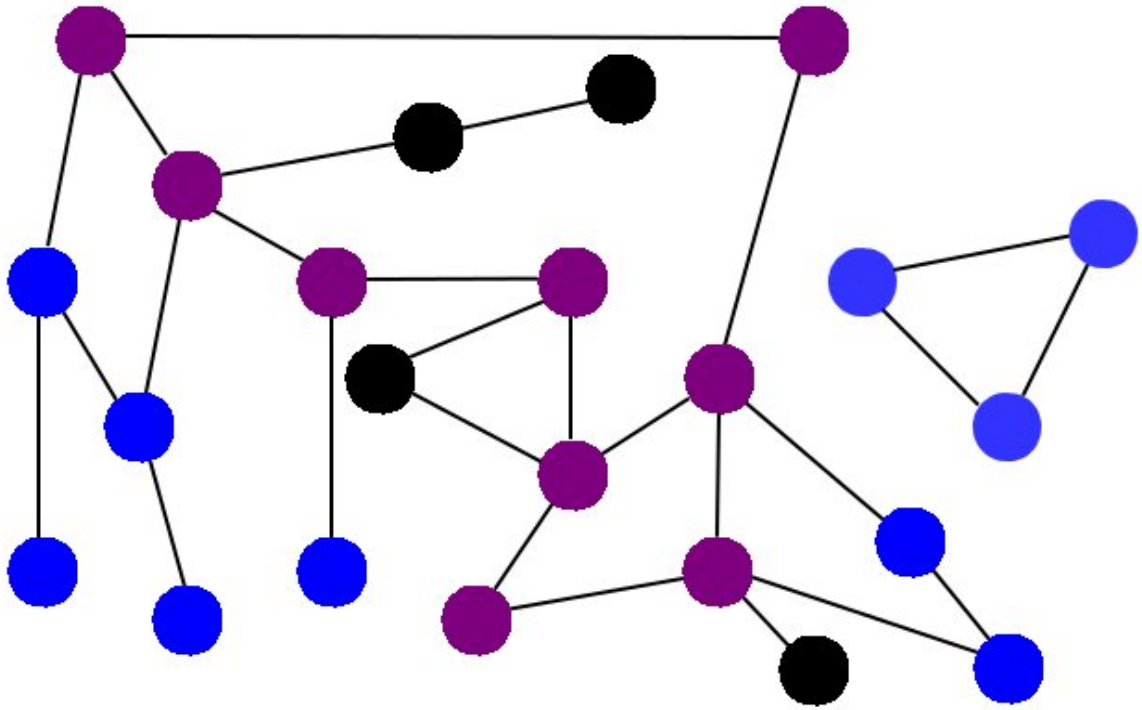
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



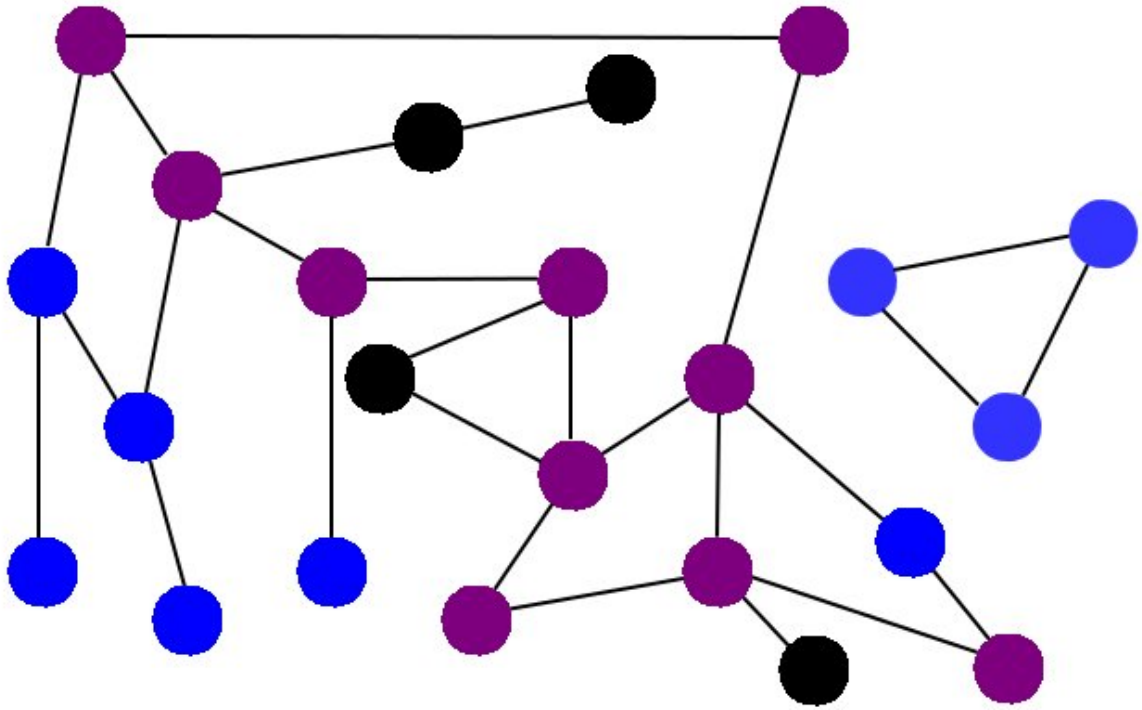
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



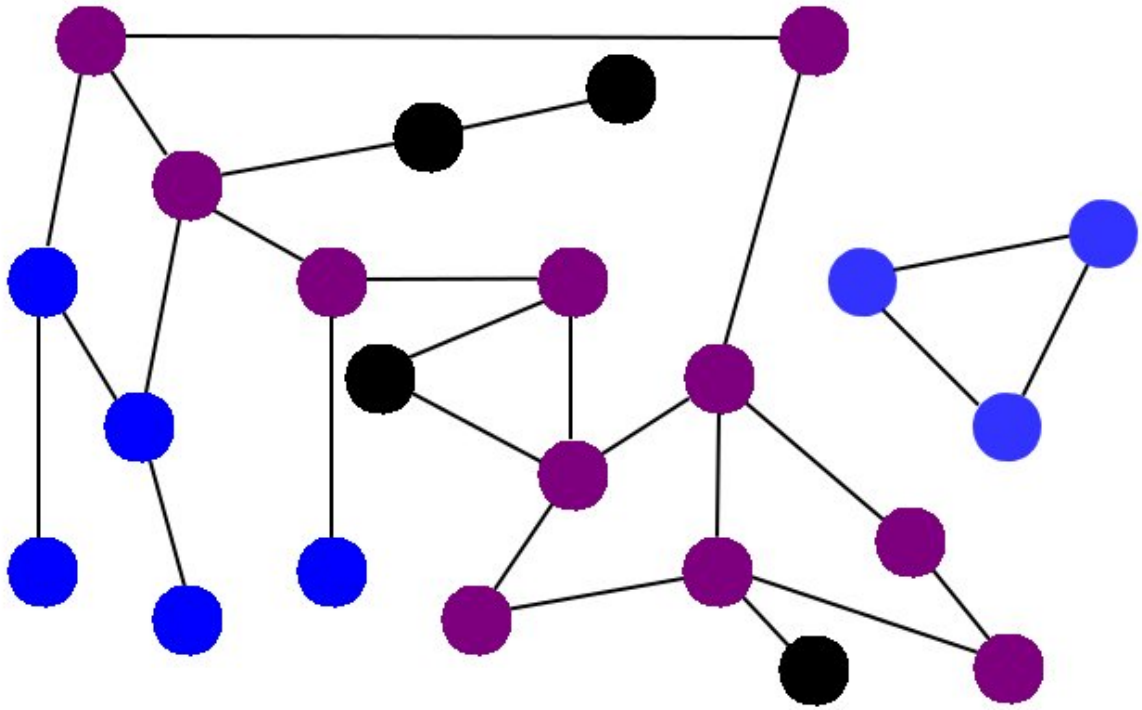
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



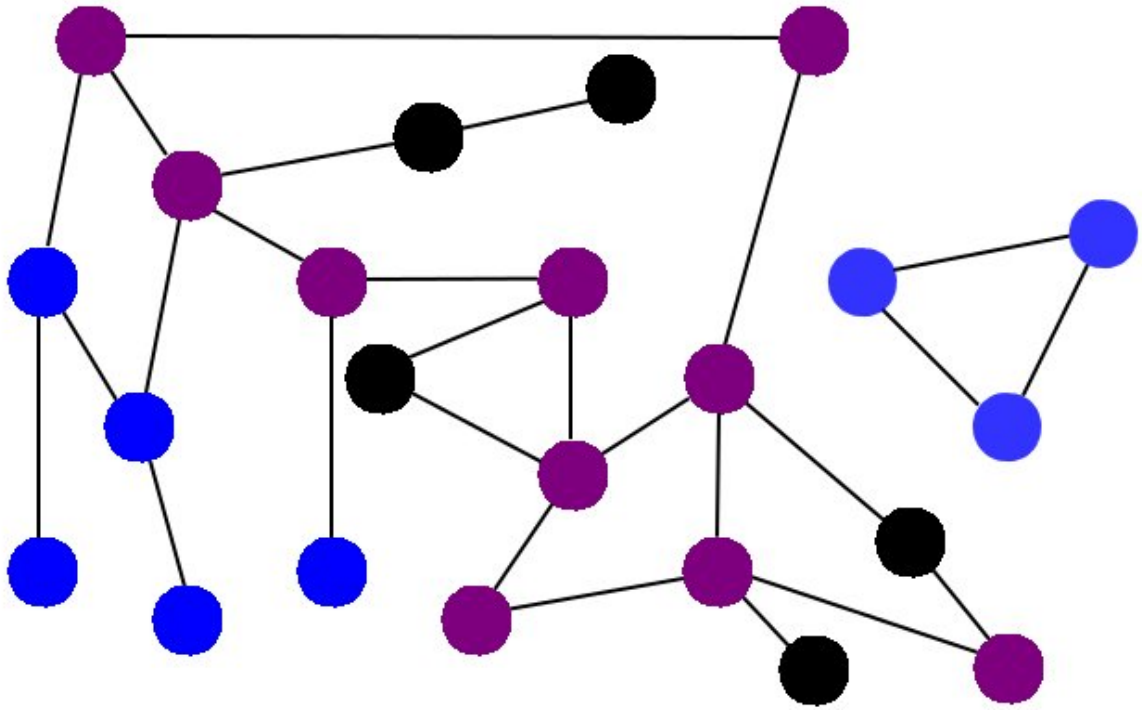
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



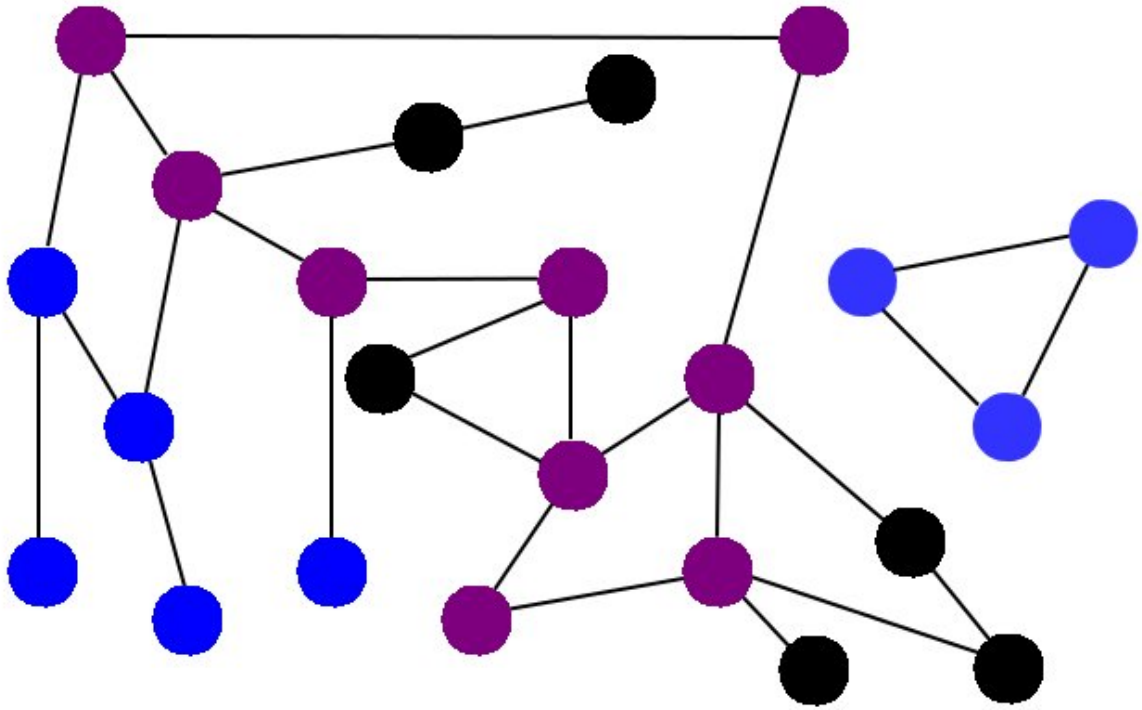
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



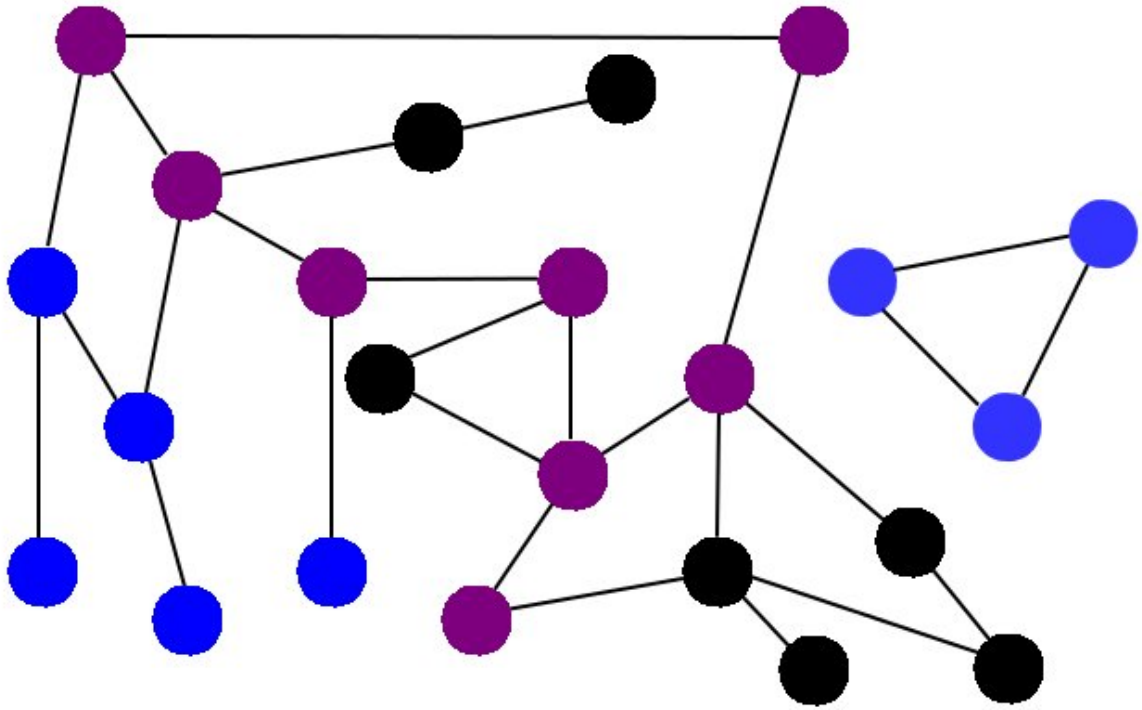
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



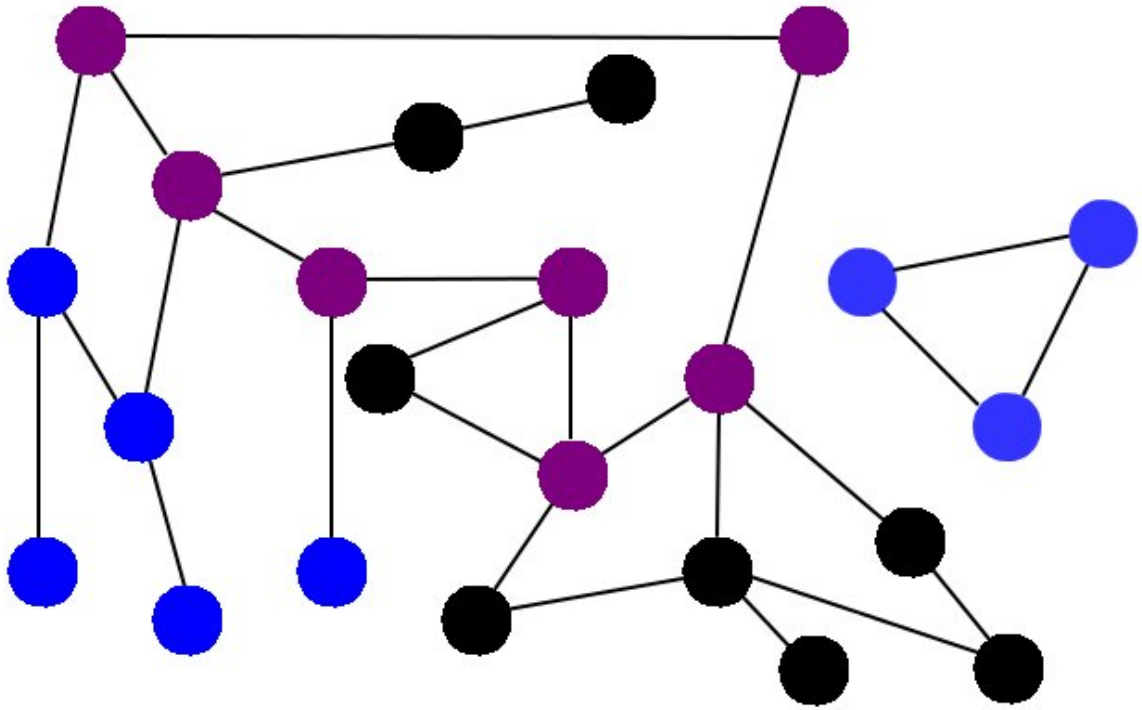
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



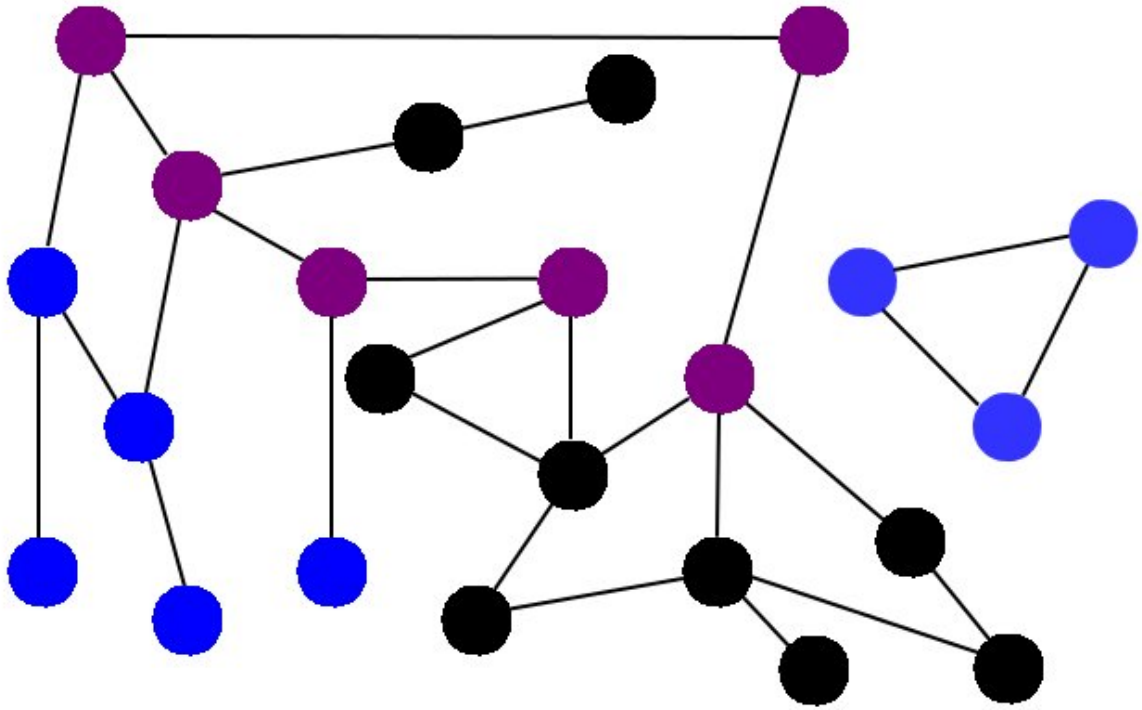
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



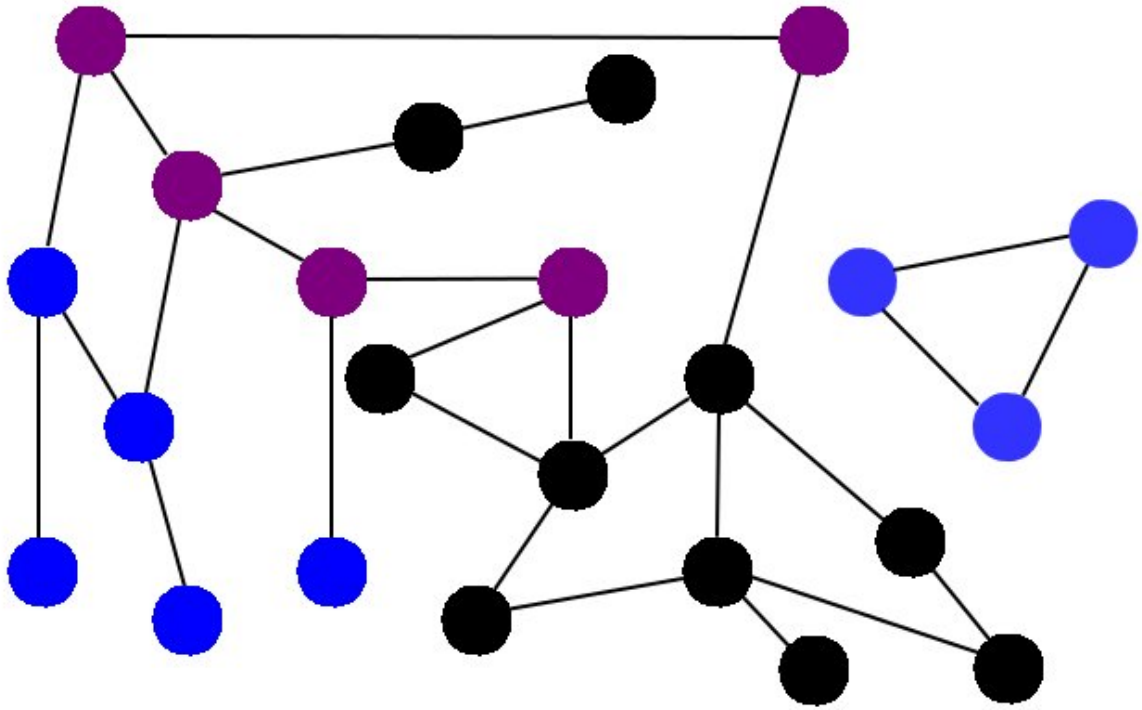
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



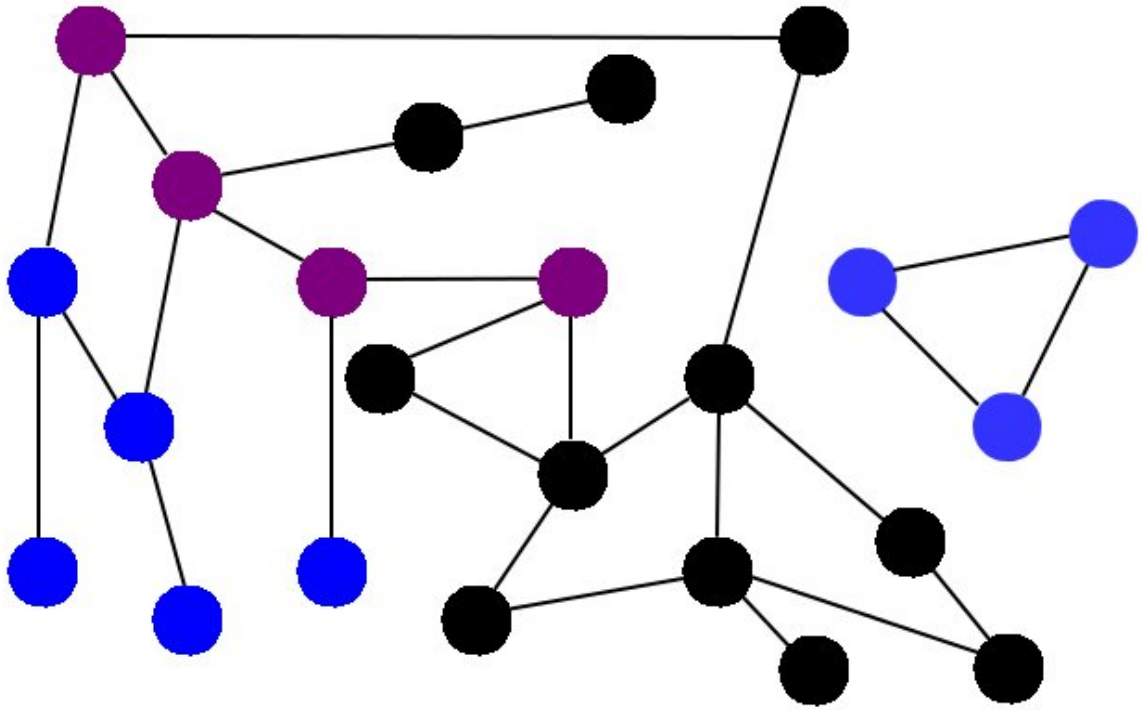
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



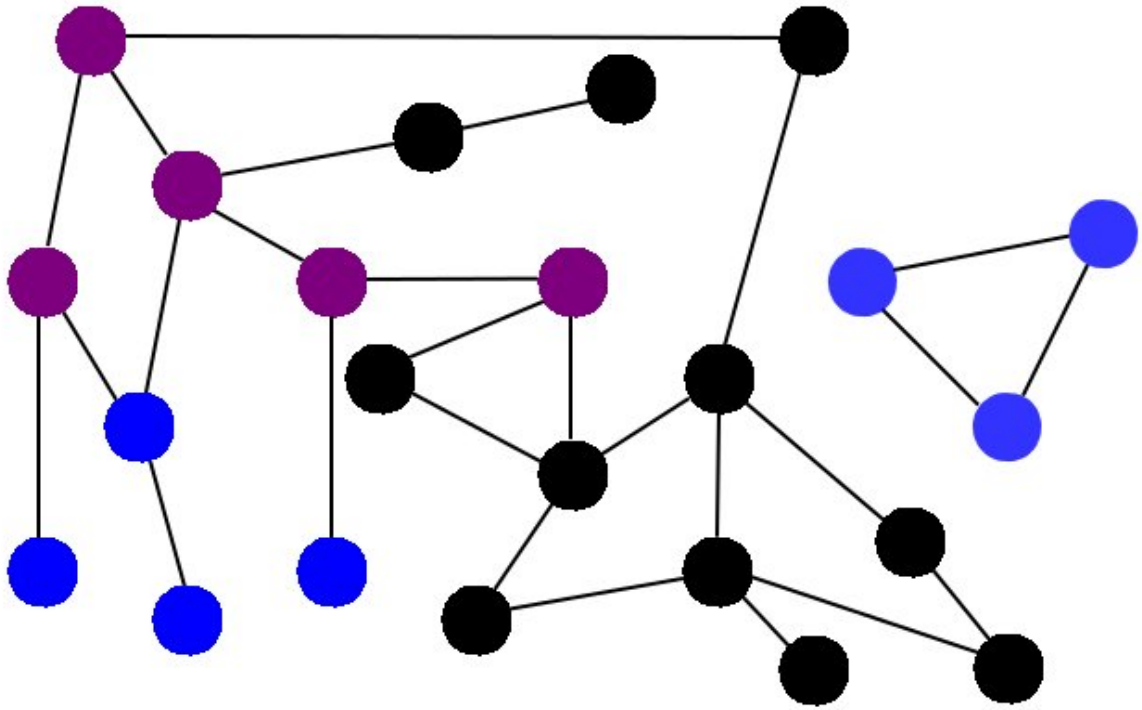
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



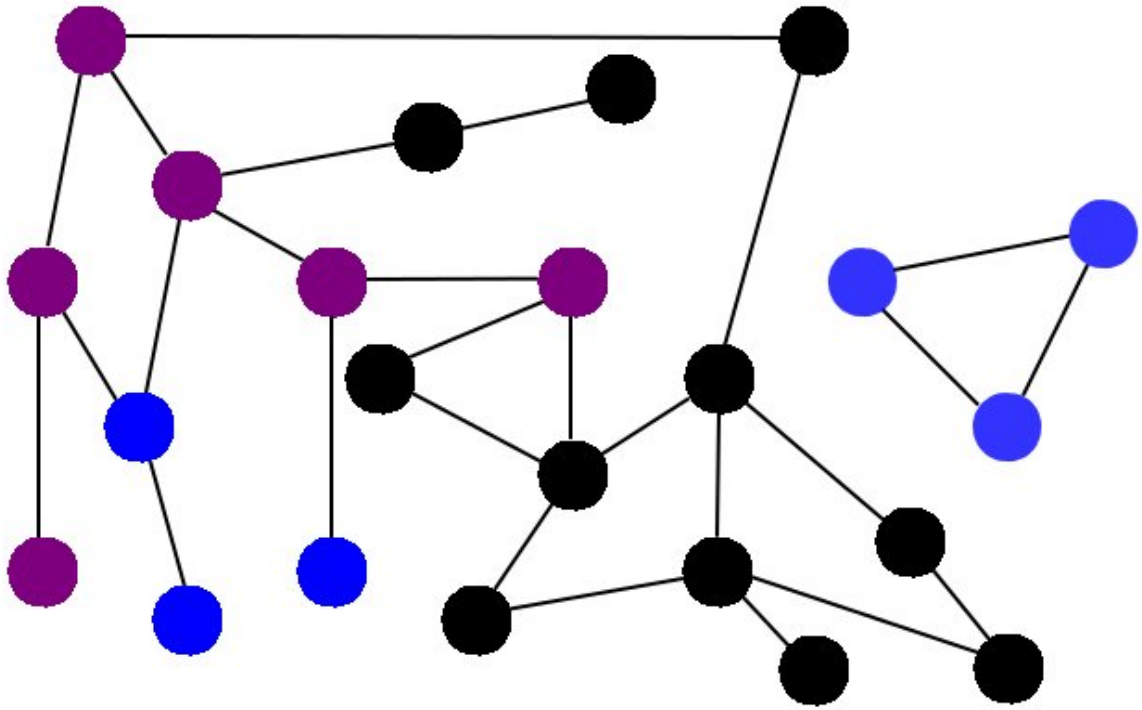
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



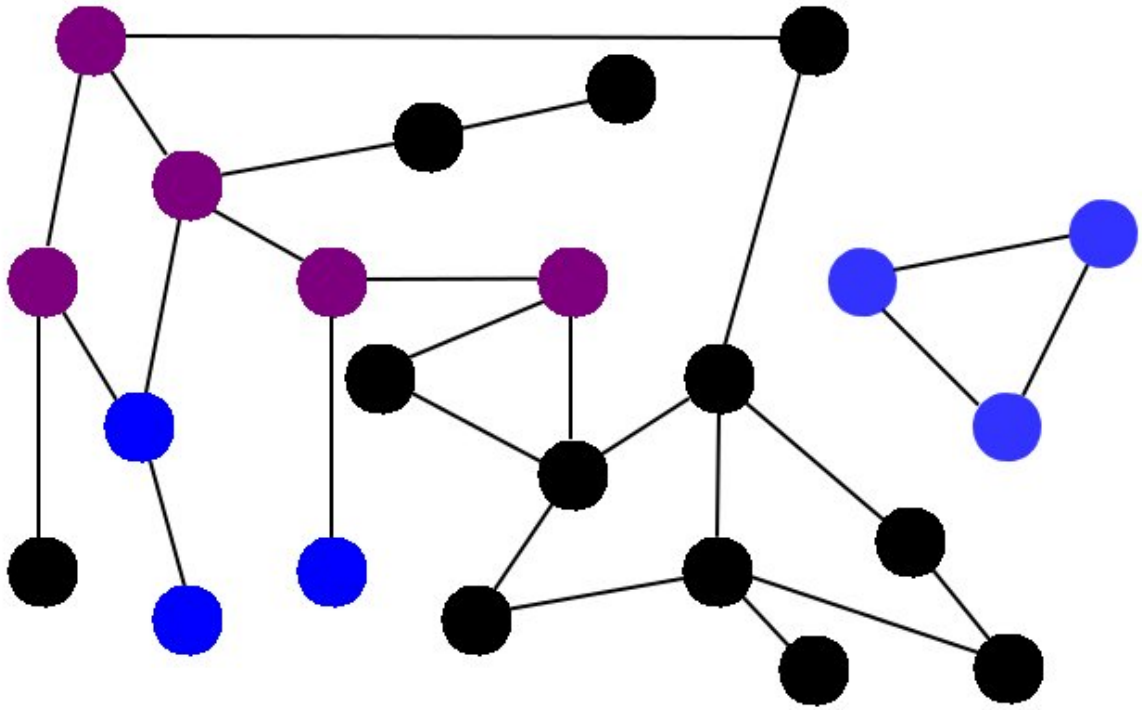
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



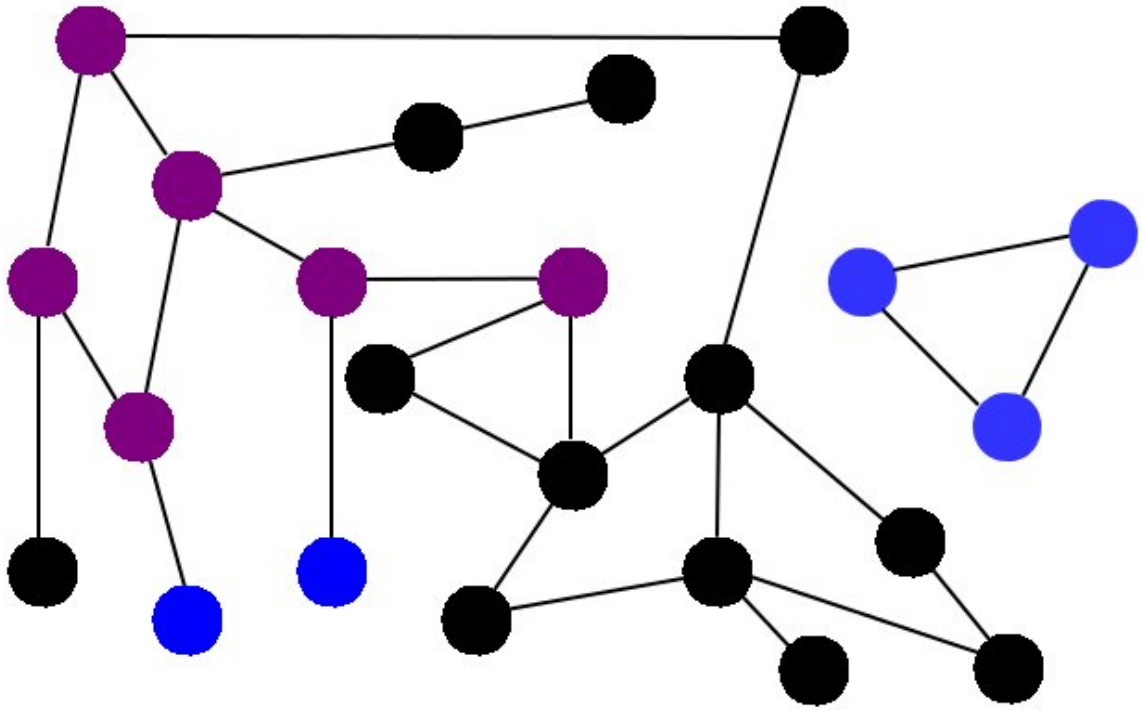
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



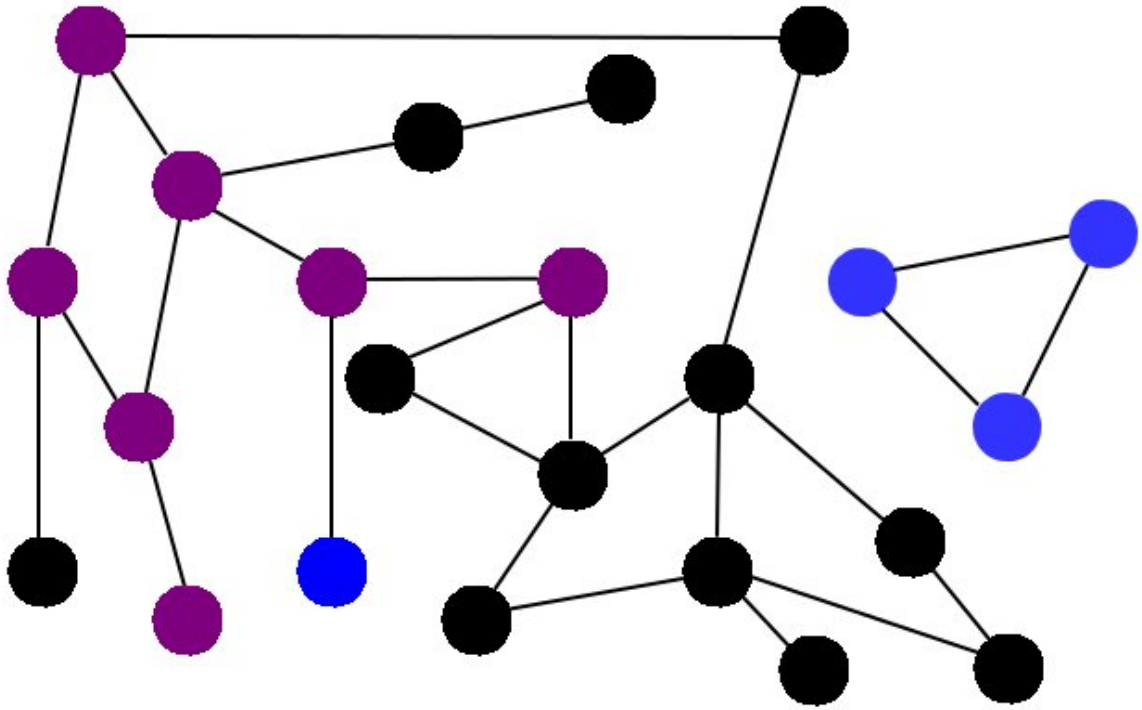
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



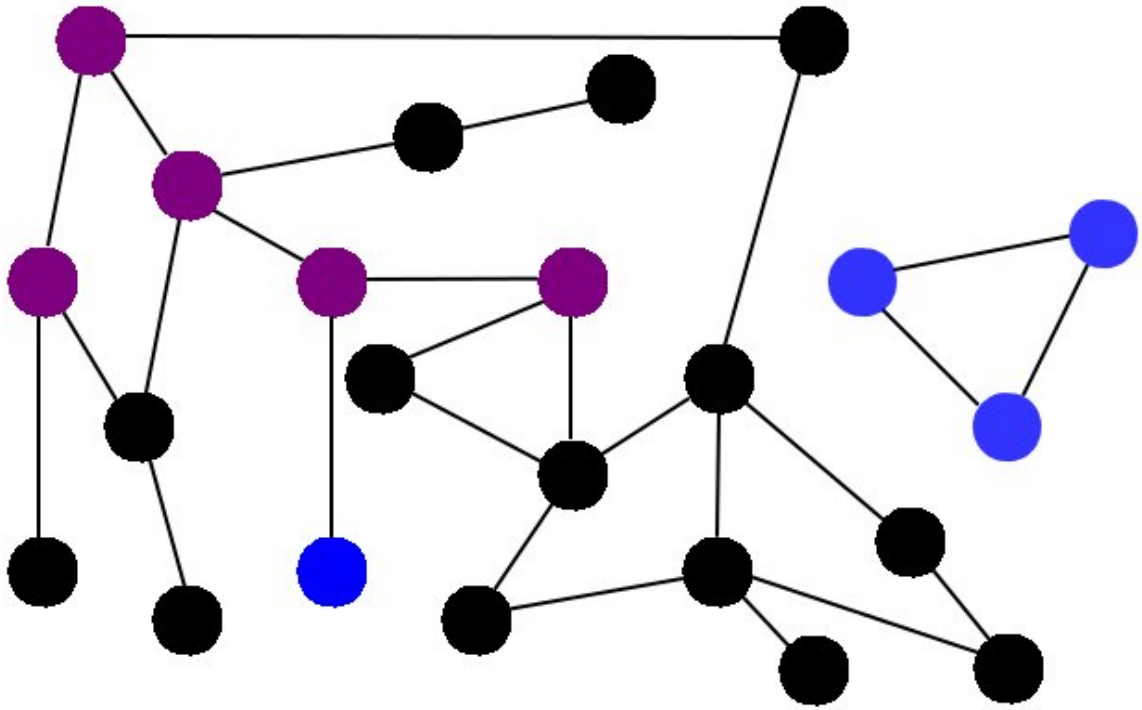
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



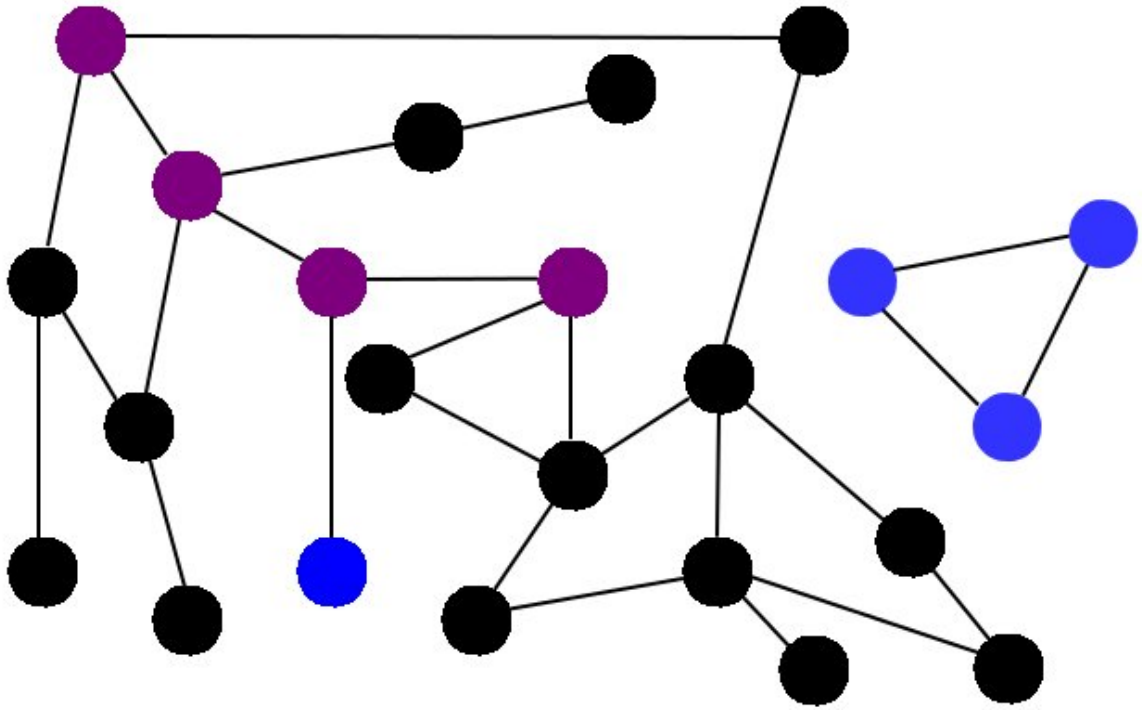
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



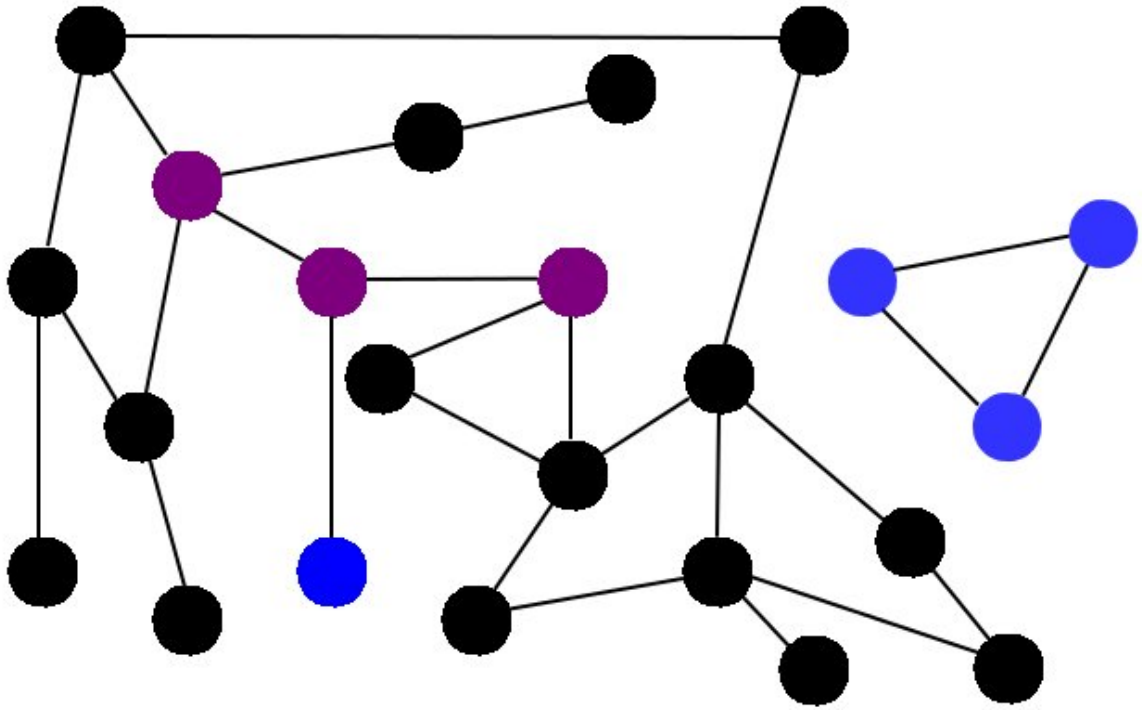
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



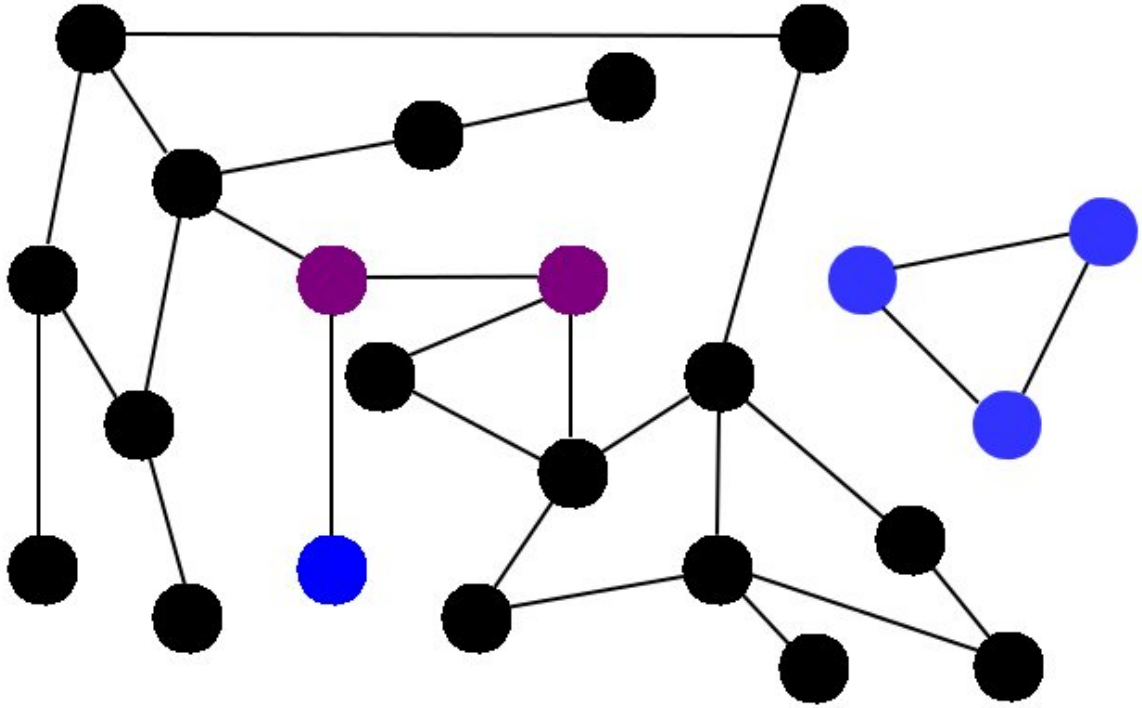
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



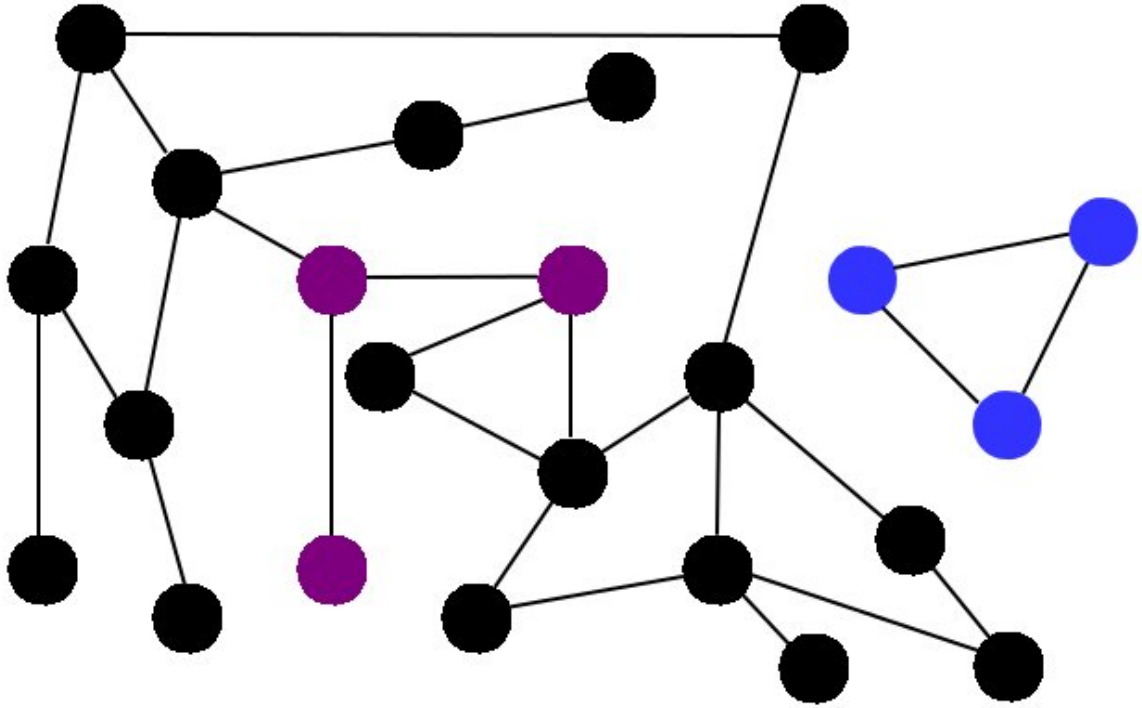
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



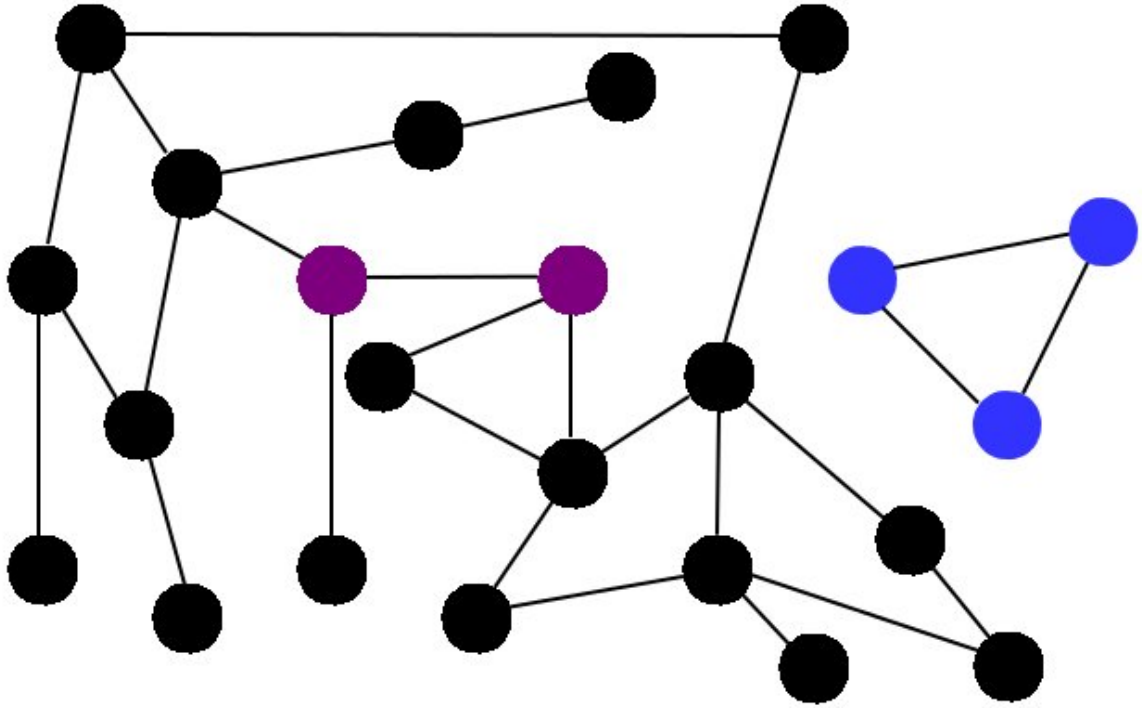
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



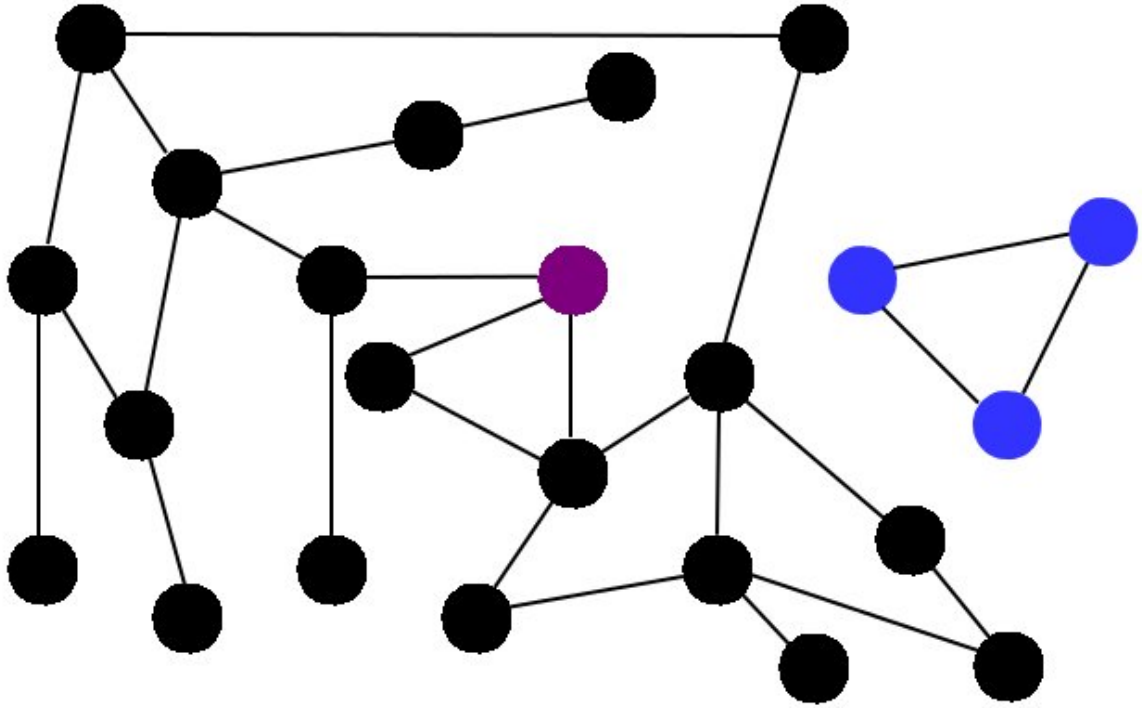
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



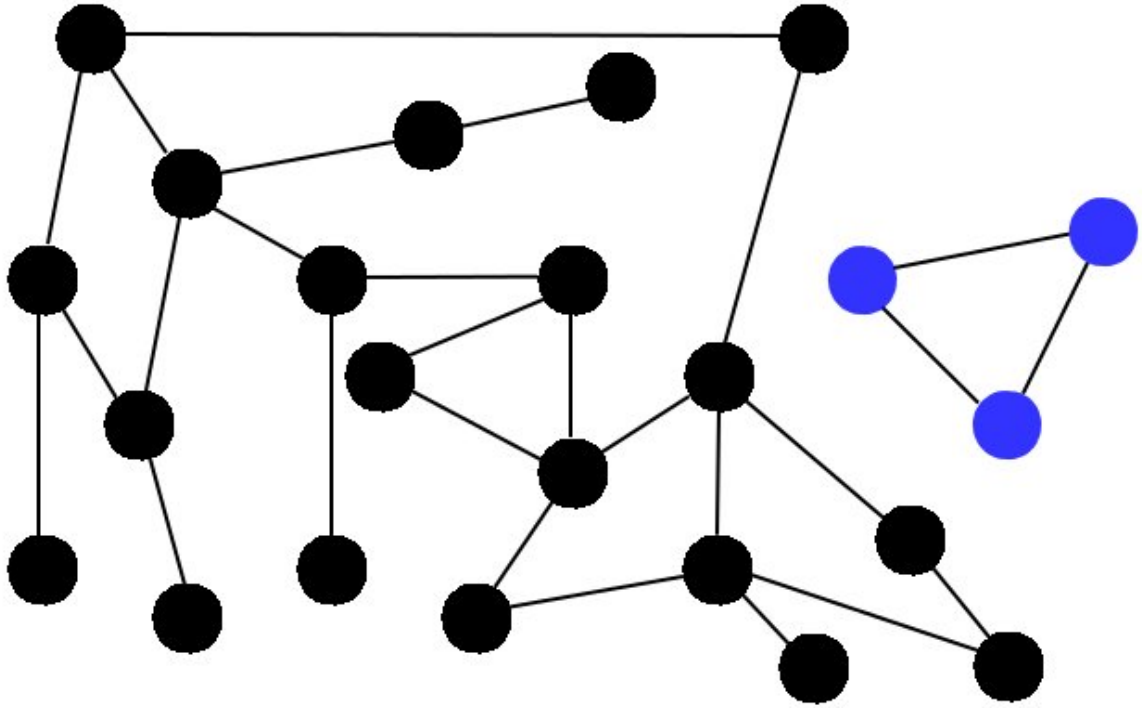
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



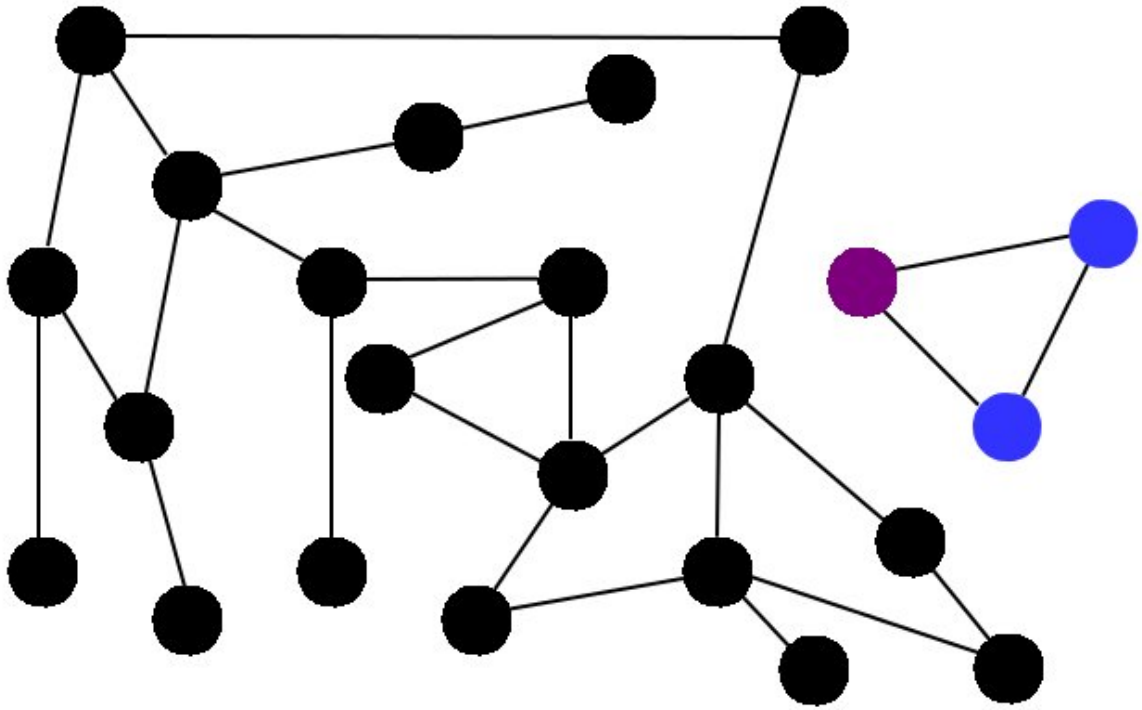
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



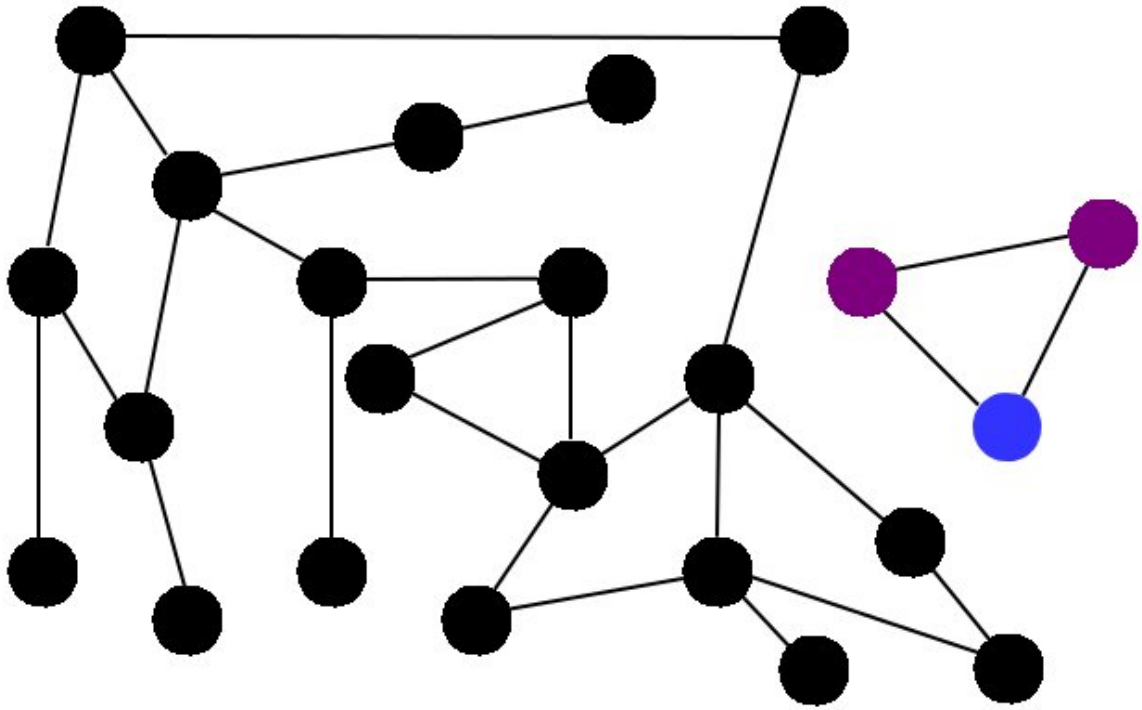
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



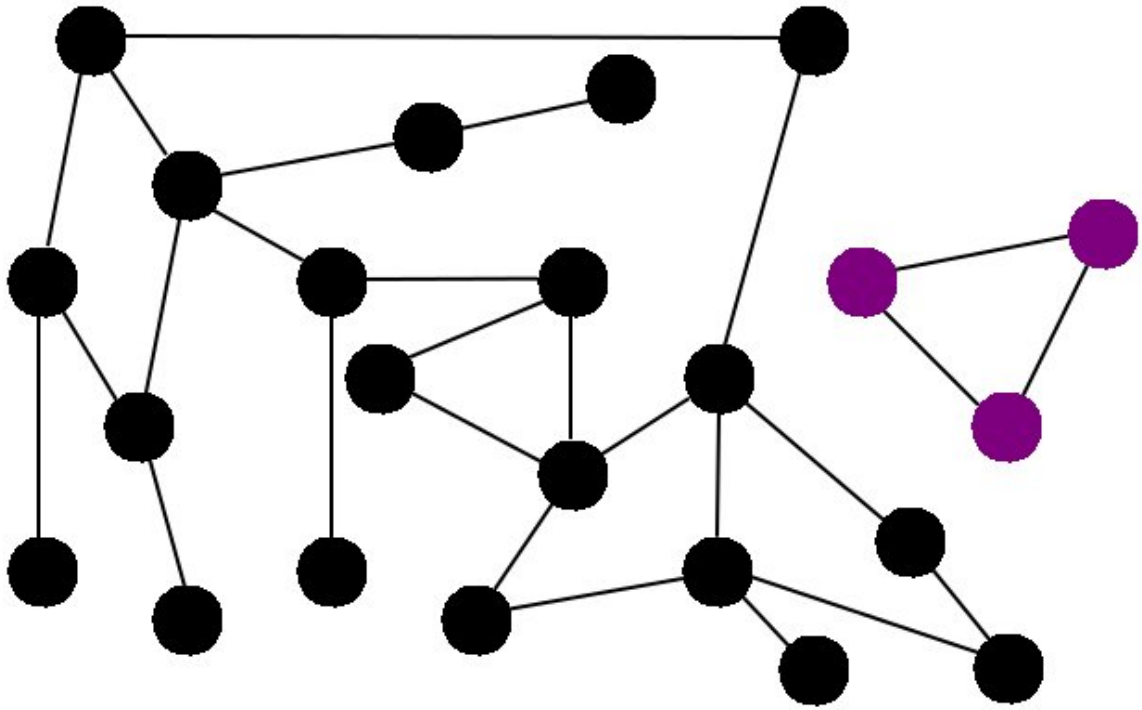
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



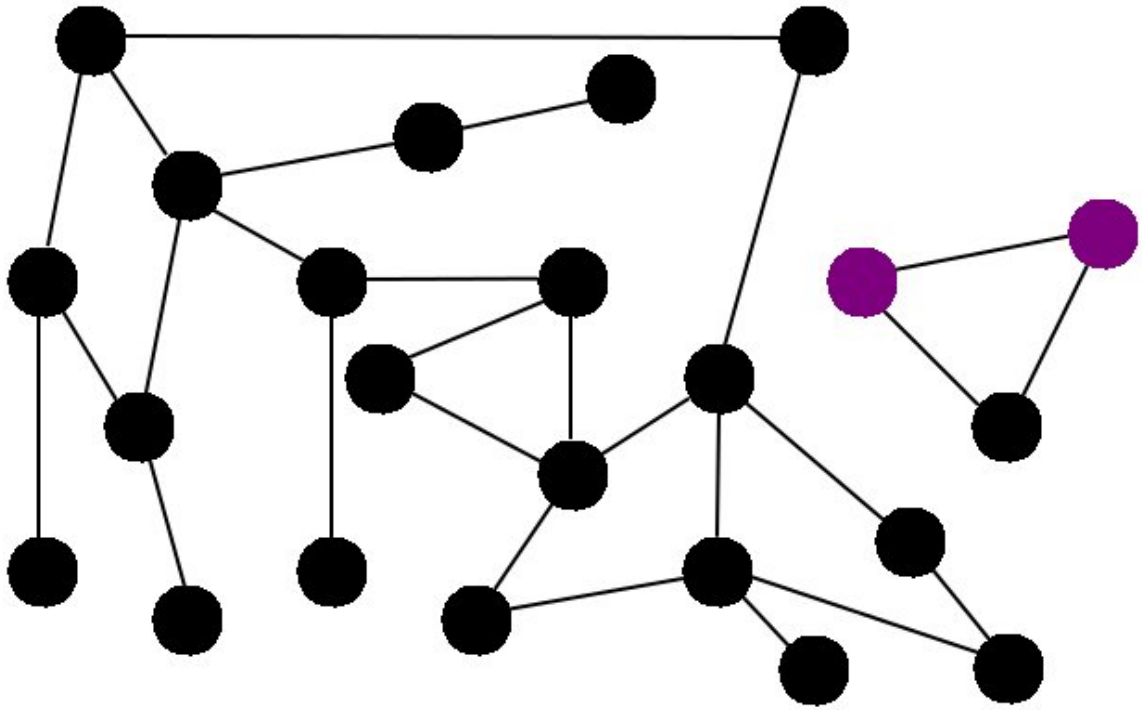
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



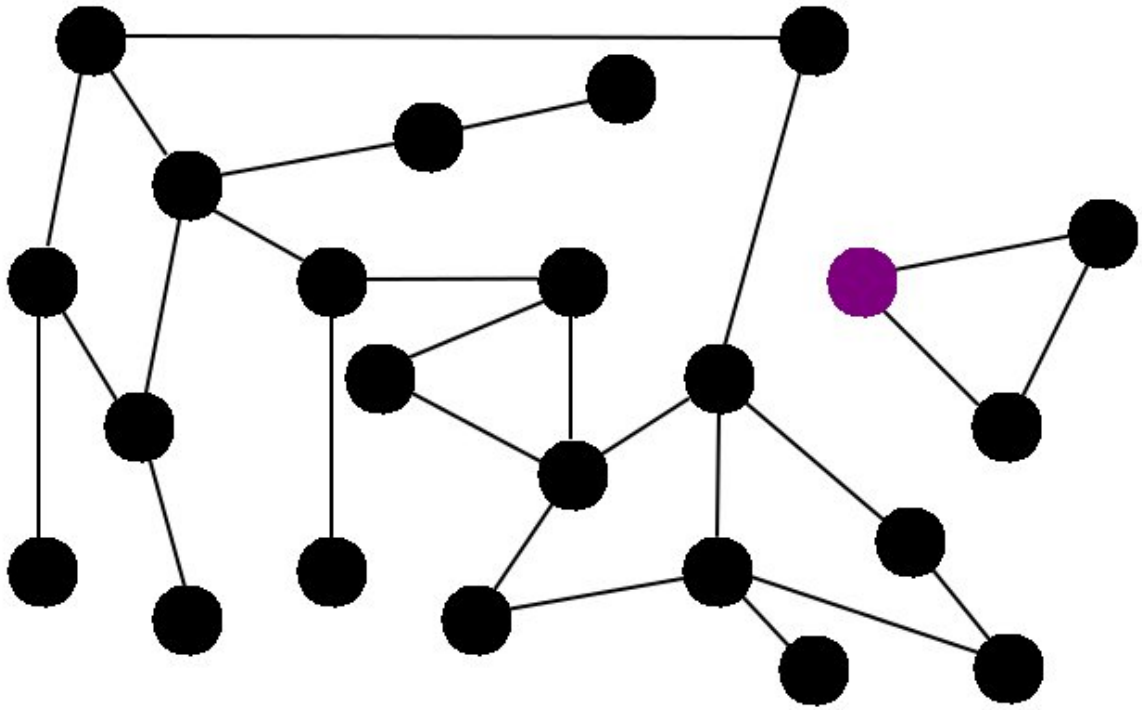
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



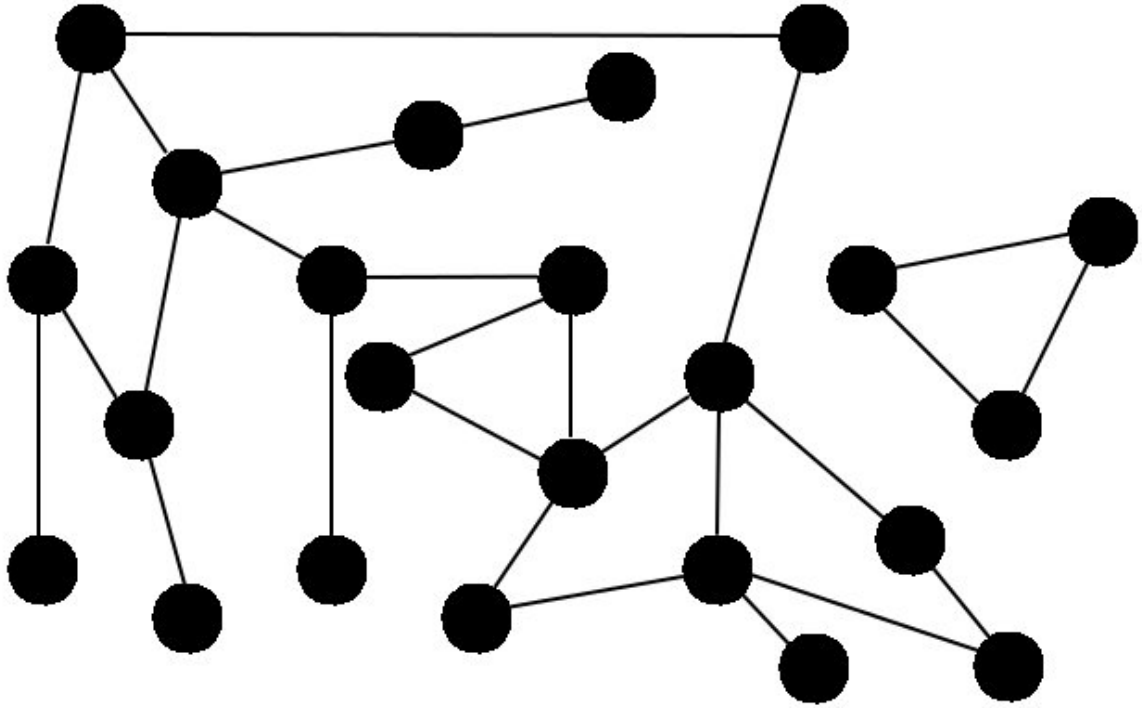
Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



Depth-First Search

A depth-first search follows a single path as far as possible and then backtracks to the last alternative path



Depth-First Search

DFS (Given graph g and source vertex s)

- Mark s as visited
- For each vertex v such that g has an edge $\{s, v\}$
  - If v has not yet been visited
    - DFS(g,v)
- HALT

- DFS does not require explicit storage of vertices like BFS
- Instead, calls for each vertex build up on the execution stack
- A loop and an explicit stack could be used
- The visited vertices must be tracked across multiple calls

Implementing DFS

- This class finds a cycle in a graph using DFS search
- The search results are stored in the `parent` and `inCycle` arrays
- All components are searched

```
1   public class DFSCycle {
2
3       private static final int FIRST = 0;
4       private static final int  LAST = 1;
5
6       private Graph     graph;
7       private int[]     inCycle;
8       private boolean[] visited;
9       private int[]     parent;
10
11      public DFSCycle(Graph g) {
12          graph   = g;
13          int v   = g.vertices();
14          visited = new boolean[v];
15          parent  = new int[v];
16          Arrays.fill(parent, -1);
17          int i = 0;
18          while (inCycle == null && i < v) {
19              if (!visited[i]) { dfsCycle(g, i); }
20              i++;
21          }
22      }
23
24      ...
```

Implementing DFS

- A cycle exists if `v` was already visited, unless it is `u`'s parent
- Since `u` was just visited, and not directly from `v`, the search must be on some alternative path from `v` to `u`
- `u` and `v` are saved to `inCycle`

```
1       ...
2
3       private void dfsCycle(Graph g, int u) {
4           visited[u] = true;
5           for (int v : graph.adj[u]) {
6               if (inCycle == null && visited[v] && parent[u] != v) {
7                   inCycle = new int[]{v, u};
8                   return;
9               }
10              if (!visited[v]) {
11                  parent[v] = u;
12                  dfsCycle(graph, v);
13              }
14          }
15      }
16
17      ...
```

Implementing DFS

- `getCycle()` returns the list of vertices within the cycle
- The path to the final cycle vertex is traced back to the first

```
1       ...
2
3       public boolean hasCycle() {
4           return inCycle != null;
5       }
6
7       public List<Integer> getCycle() {
8           List<Integer> cycle = new List<Integer>();
9           int v = inCycle[LAST];
10          cycle.add(v);
11          do {
12              cycle.add(parent[v]);
13              v = parent[v];
14          } while (v != inCycle[FIRST]);
15
16          return cycle;
17      }
18  }
```

DFS Performance

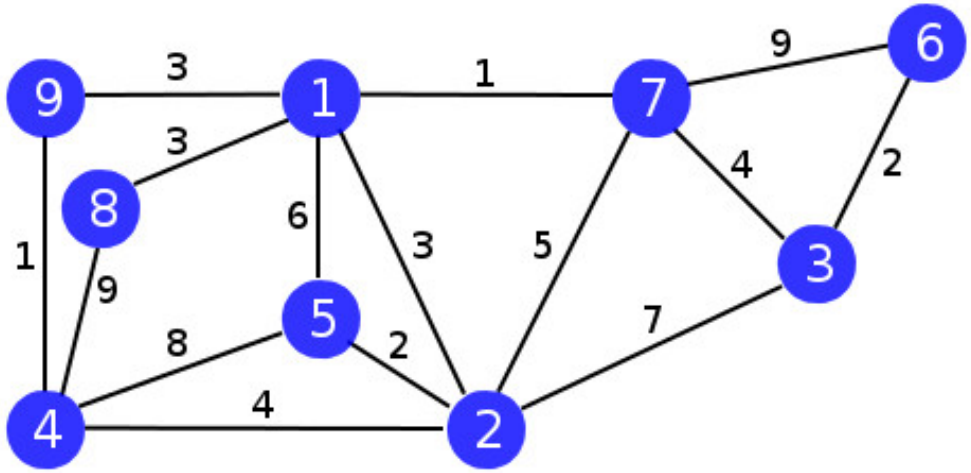
For a graph with V vertices and E edges DFS also runs in $O(V + E)$ time

- DFS is called exactly once per vertex
- Each adjacency list is used exactly once
- Each edge contributes exactly two vertices to the adjacency lists

Part III

Minimum Spanning Trees

More Terminology

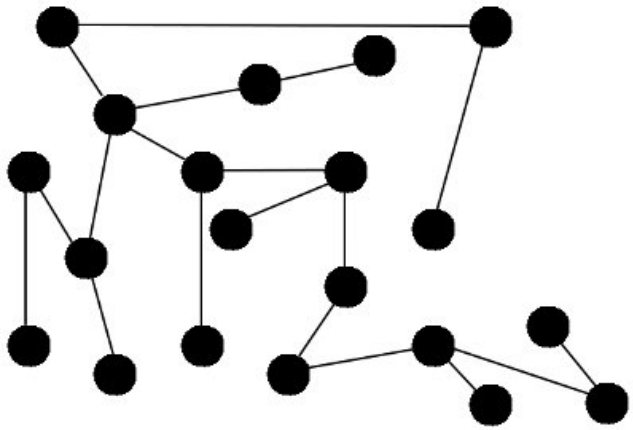


- Each edge in a weighted graph has an associated cost or weight
- We denote the weight of the edge $\{u, v\}$ by $w(u, v)$

More Terminology

Definition (Tree)

A tree is a pair (G, r) where G is a connected, acyclic graph and r is a vertex of G, called the root.

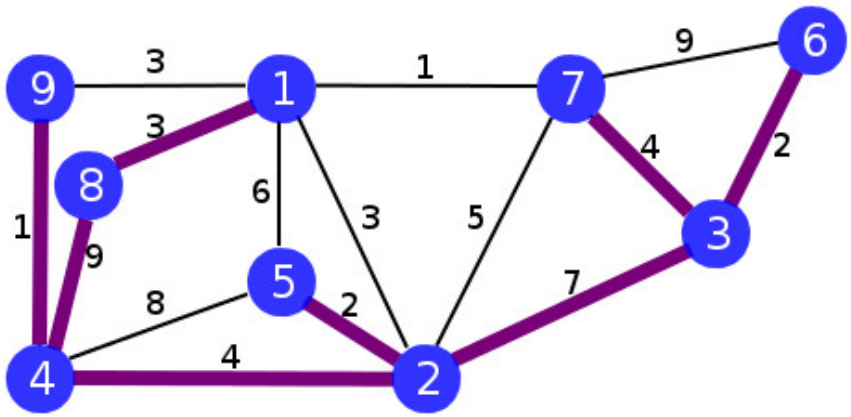


- A nonrooted tree is a connected, acyclic graph

More Terminology

Definition (Spanning Tree)

Given a graph $G = (V, E_G)$, a tree $T = (V, E_T)$ such that $E_T \subseteq E_G$ is a spanning tree for G.



Minimum Spanning Tree

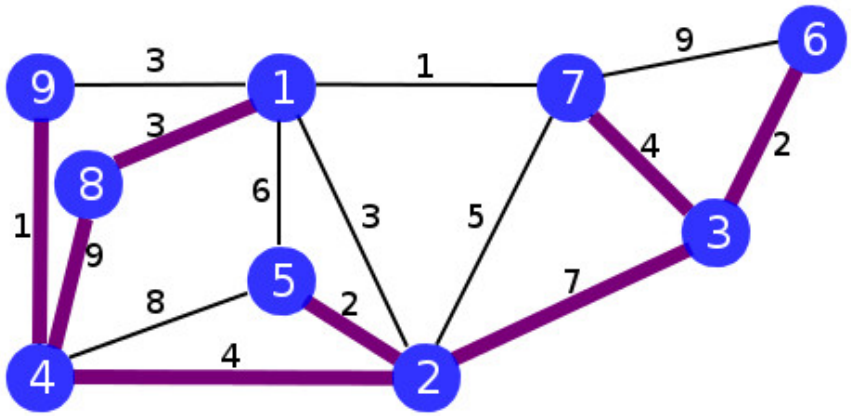
The Minimum Spanning Tree Problem

Input Weighted graph G

Output Spanning tree T for G such that the sum

$$w(T) = \sum_{\{u,v\} \in T} w(u,v)$$

is minimised (there is no T' where $w(T') < w(T)$).



A New Strategy

There are lots of possible spanning trees for a given graph

- We could find each one and compute its weight

Or we could see if a greedy approach might work

- A greedy algorithm picks the 'obvious' first step
- This is called making a greedy choice
- It leaves a subproblem to solve

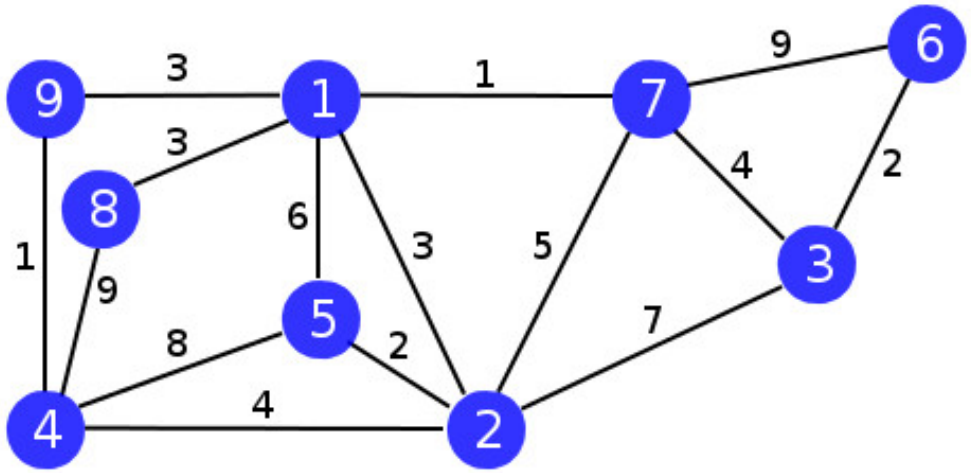
The Greedy Choice

The Greedy Choice

The Greedy Choice

The Greedy Choice

What greedy choices are there when computing an MST?



The Greedy Choice

- Greed is only good sometimes
- We have to show that the choice must lead to a correct solution

Theorem

Let G be a connected, weighted graph. If e_m is an edge of least weight in G, then e_m is in some minimum spanning tree for G.

The general method of proving that the greedy choice is OK is:

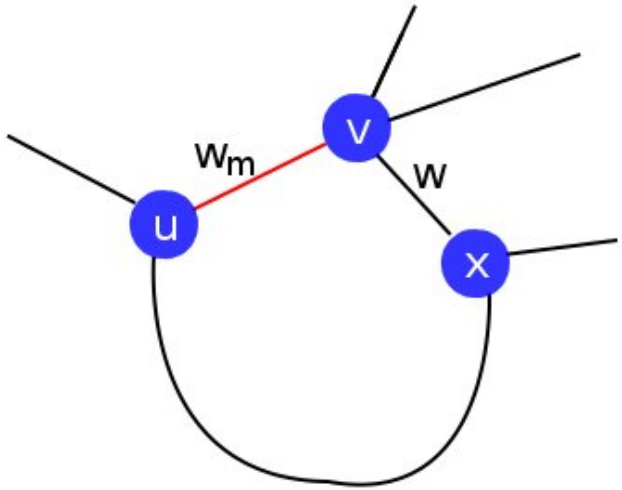
1. Suppose you have an optimal solution to the problem
2. Show that it is still optimal when the greedy choice is included

Proof

T is some MST for G

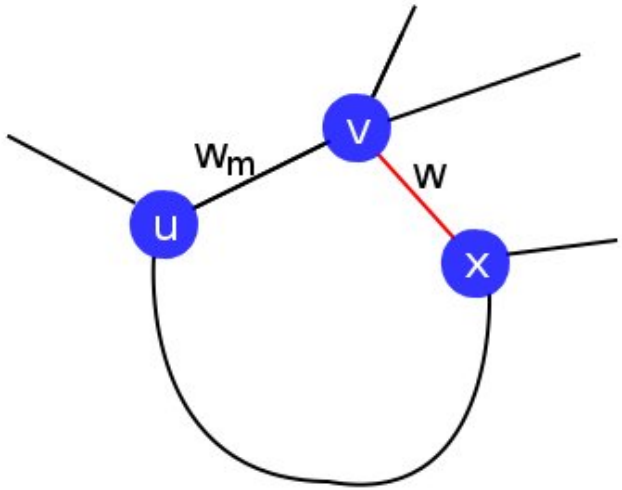
- If e_m is in T, the theorem is true
- Suppose e_m is not in T

Let $e_m = \{u, v\}$, let the path from u to v in T include the edge $\{v, x\}$, and let the weights of $\{u, v\}$ and $\{v, x\}$ be w_m and w



Proof

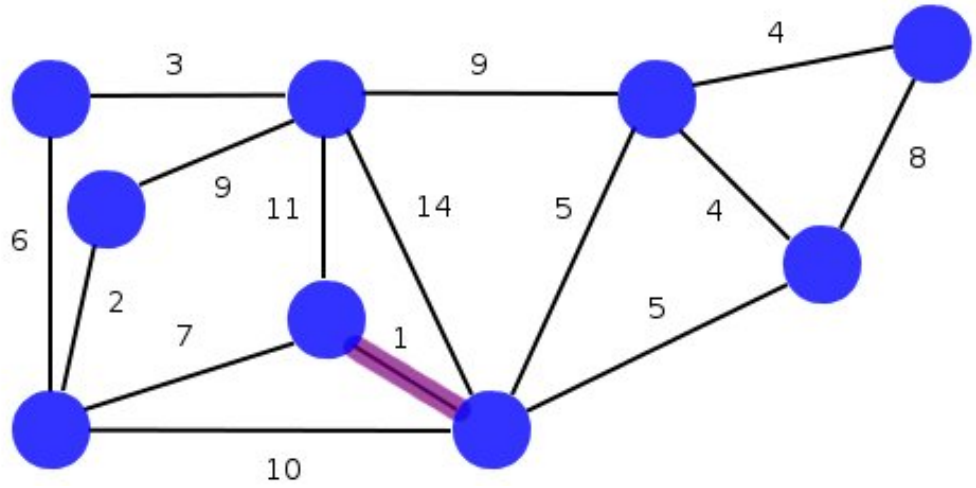
Now construct T' by removing $\{v, x\}$ from T and adding $\{u, v\}$



- Since T is a spanning tree, T' is a spanning tree
- Since T is an MST and $w_m \leq w$, T' is an MST
- e_m is in T'
- QED

Proof

Can we keep adding the next least weight edge?

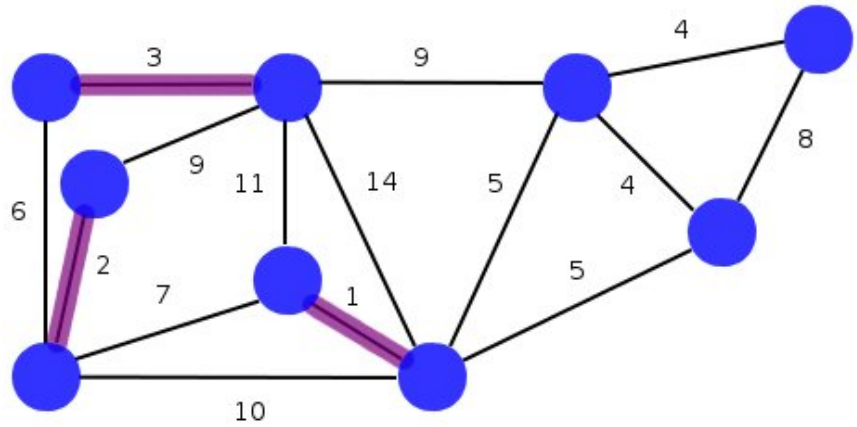


More Greed

Our greedy choice can be made more general

Theorem

Let $G = (V, E)$ be a connected, weighted graph. Let E_T be a subset of E that is part of an MST for G, and let P be a connected component in the graph (V, E_T). If E_{PQ} is the set of edges $\{u, v\}$ where exactly one of $\{u, v\}$ is in P, and e_m is an edge of least weight in E_{PQ} then e_m is in a minimum spanning tree for G.

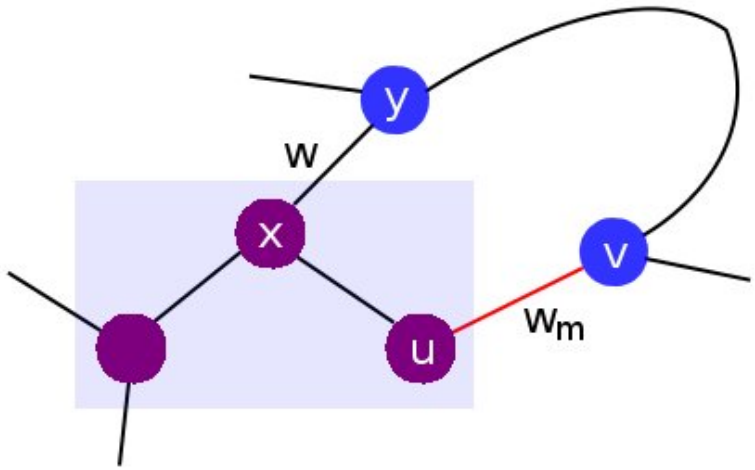


Proof

The proof is similar

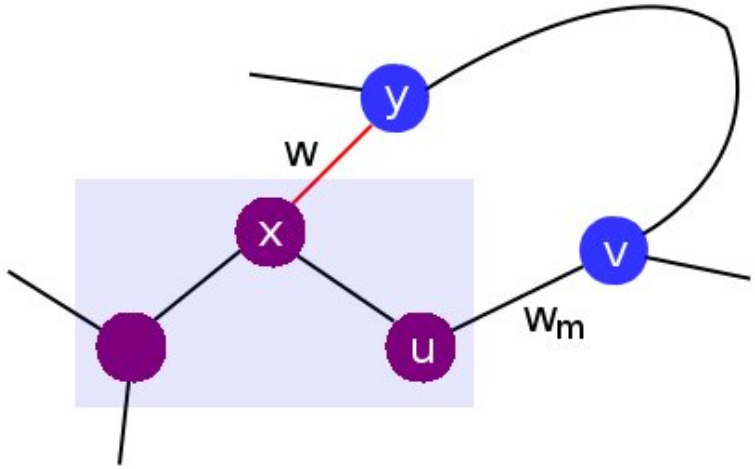
- Let T be the MST that contains E_T
- If e_m is in T, the theorem is true
- Suppose e_m is not in T

Let $e_m = \{u, v\}$, let the first edge on the path from u to v in T that is in E_{PQ} be $\{x, y\}$, and let the weights of $\{u, v\}$ and $\{x, y\}$ be w_m and w



Proof

Now construct T' by removing $\{x, y\}$ from T and adding $\{u, v\}$



- Since T is a spanning tree, T' is a spanning tree
- Since T is an MST and $w_m \leq w$, T' is an MST
- e_m is in T'
- QED

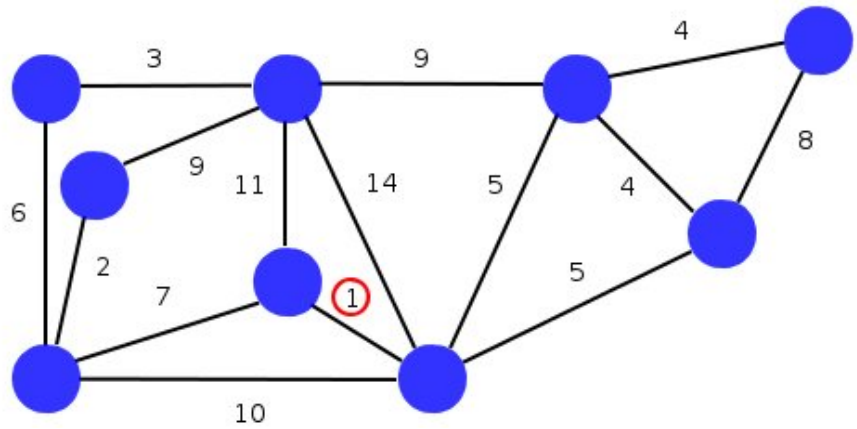
Kruskal's Algorithm

There are two well-known MST algorithms based on this greedy choice

Kruskal's Algorithm (Input: a connected, weighted graph $G = (V, E)$)

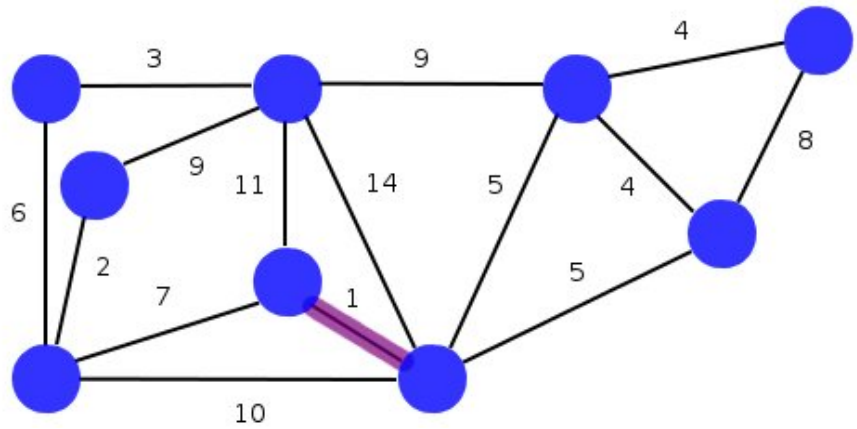
- $T = (V, \emptyset)$
- Add all edges in E to a queue Q prioritised by min weight
- For each vertex v in V
  - Make a set $S_v = \{v\}$
- While Q is not empty
  - Remove the next edge $\{x, y\}$ from Q
  - If $x \in S_i$ and $y \in S_j$ and $i \neq j$
    - Add $\{x, y\}$ to T
    - $S_i = S_i \cup S_j$
    - $S_j = \emptyset$
- Return T and HALT

Kruskal's Algorithm



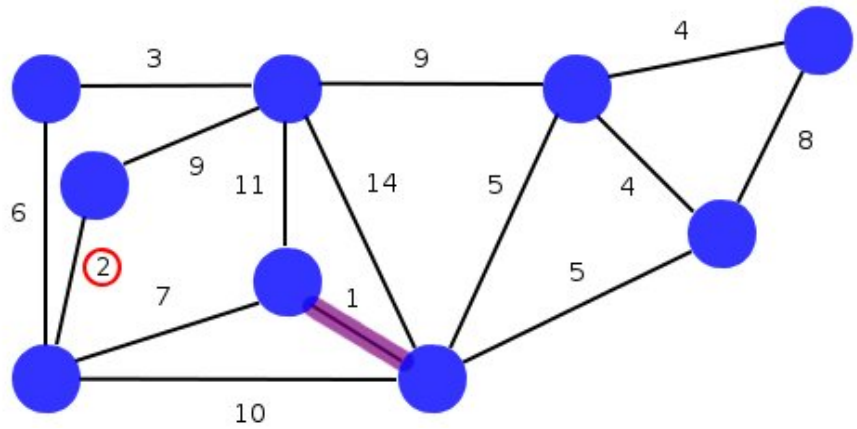
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



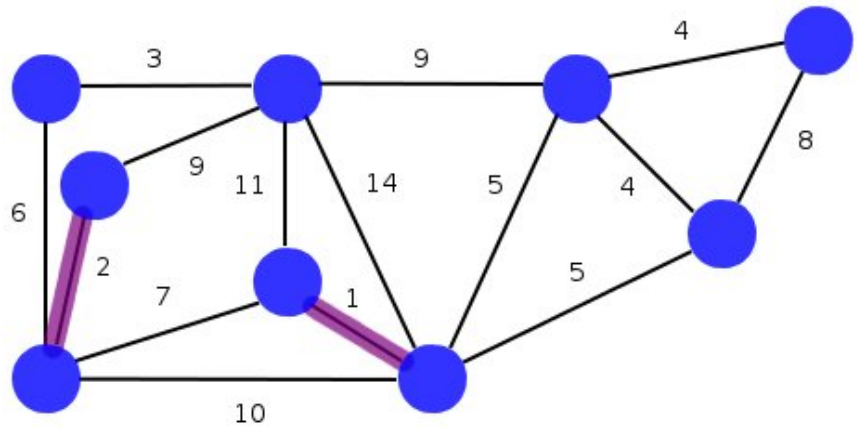
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



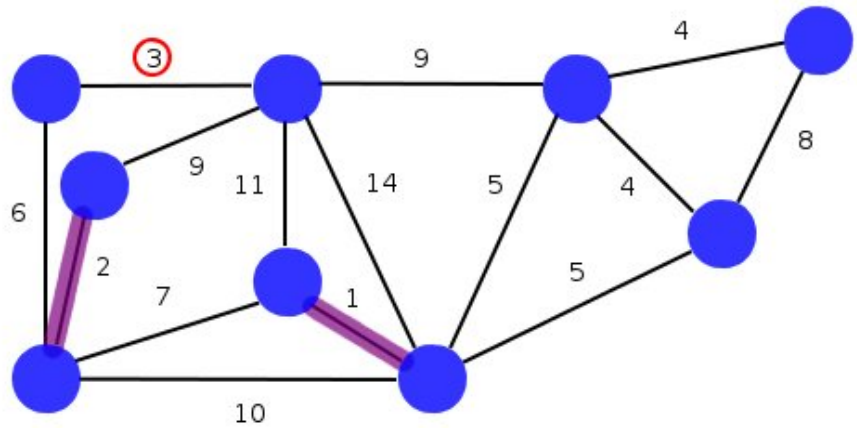
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



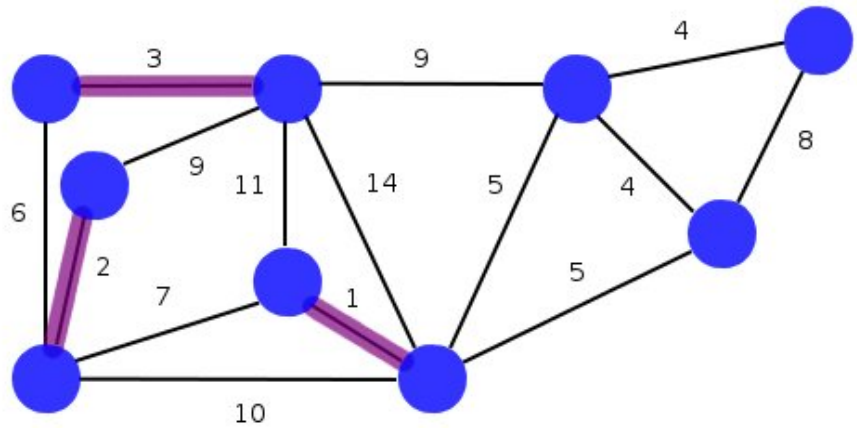
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



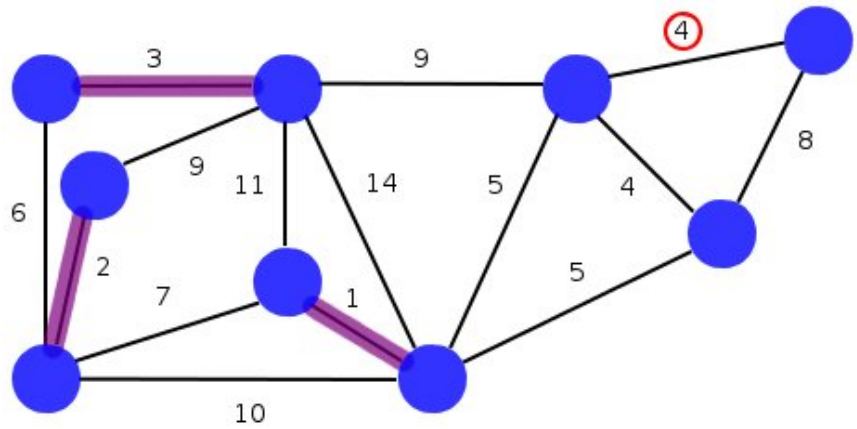
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



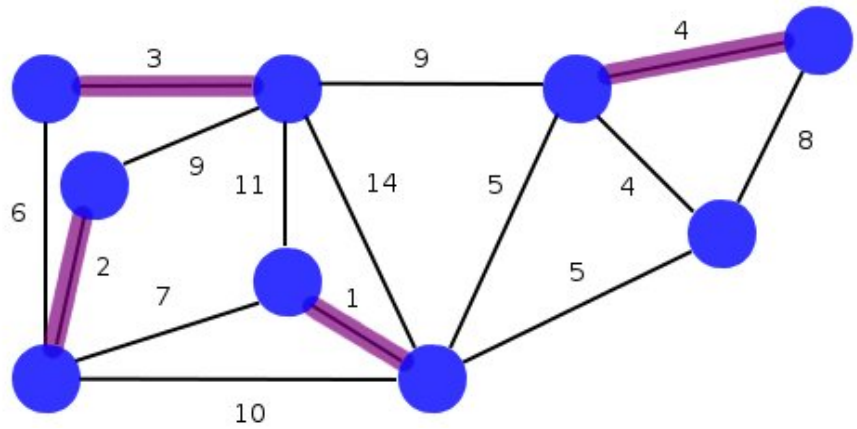
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



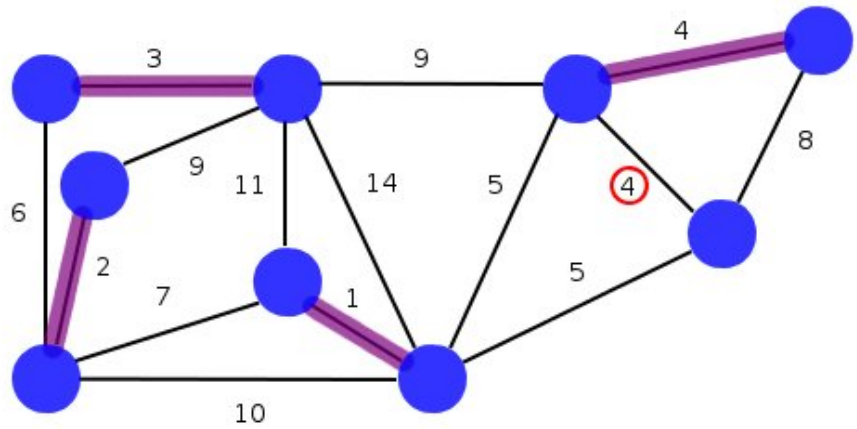
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



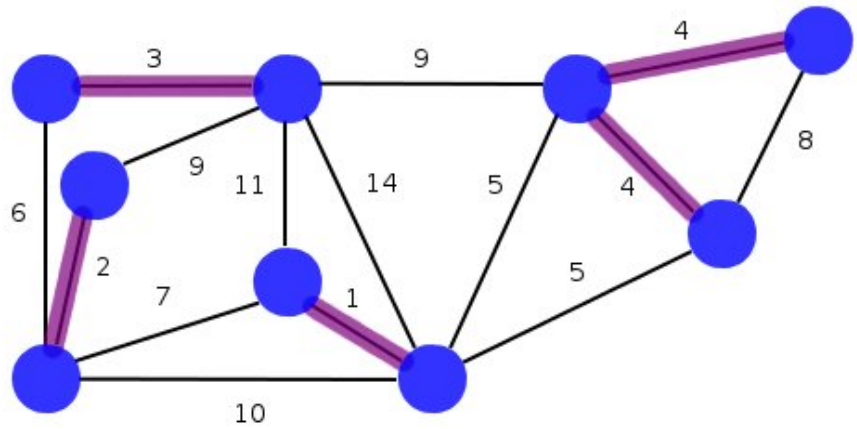
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



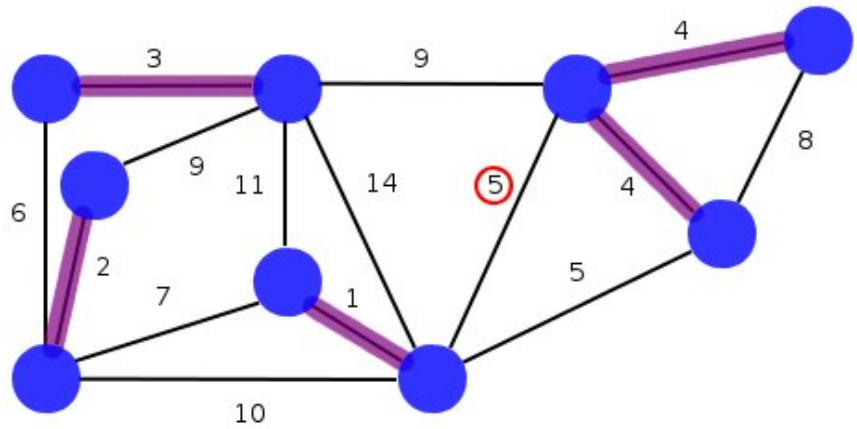
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



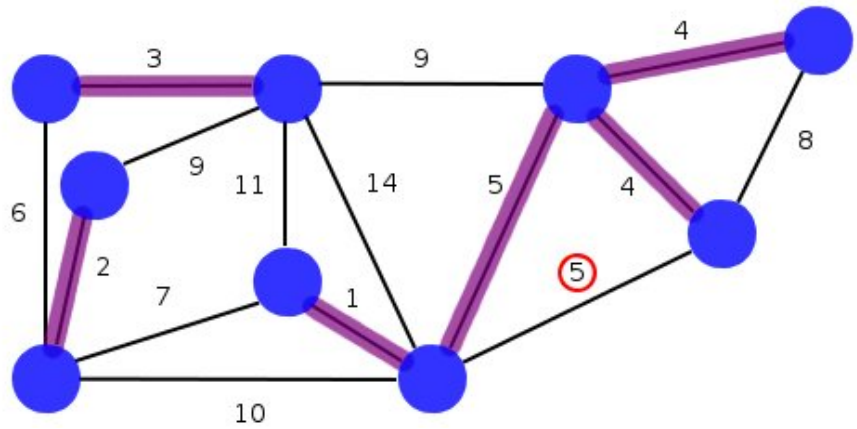
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



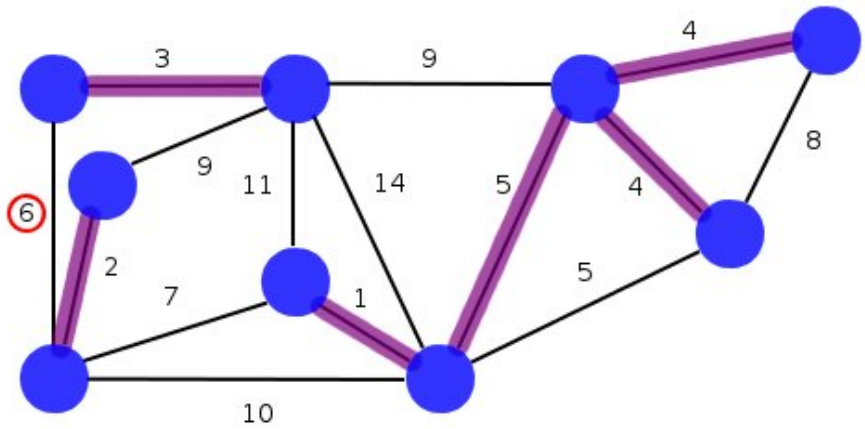
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



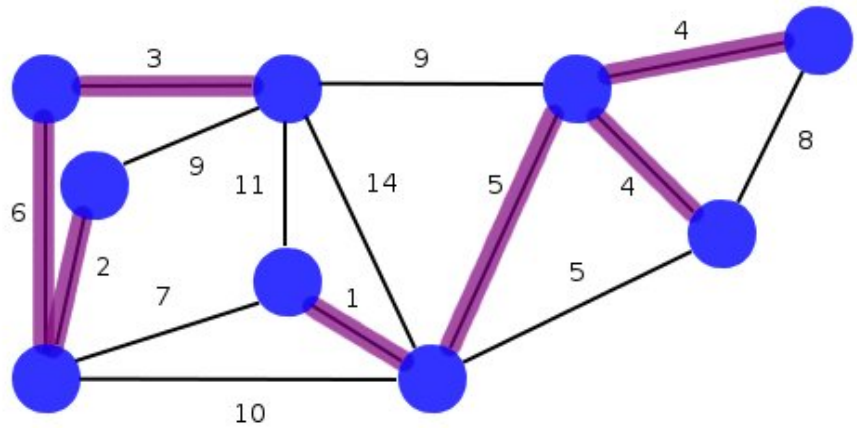
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



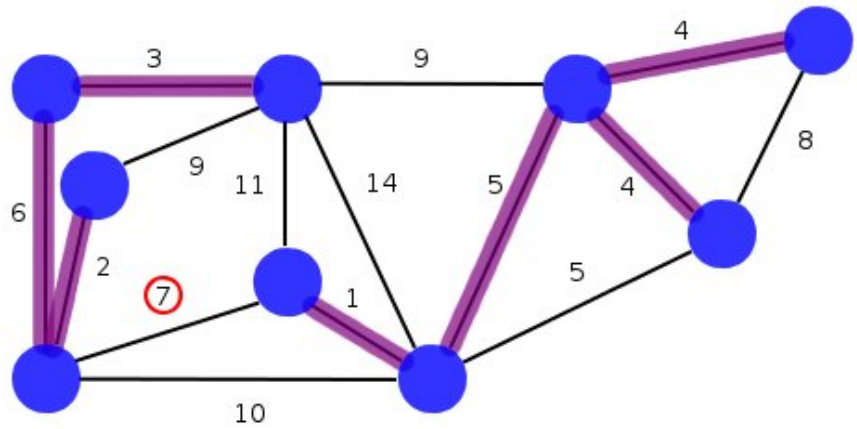
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



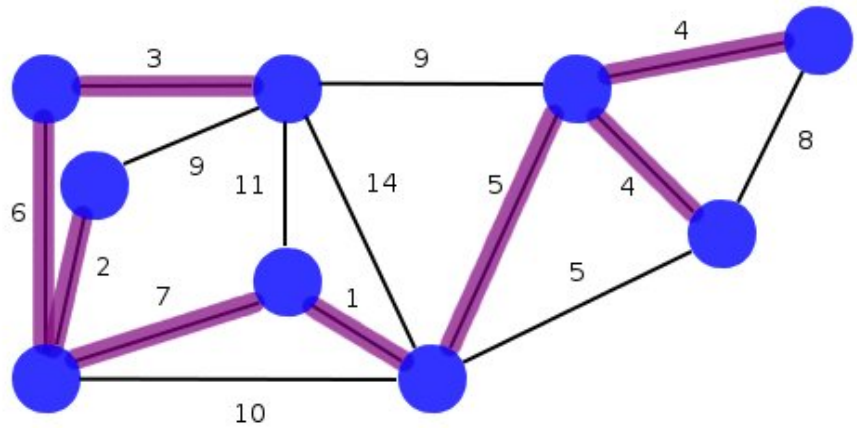
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



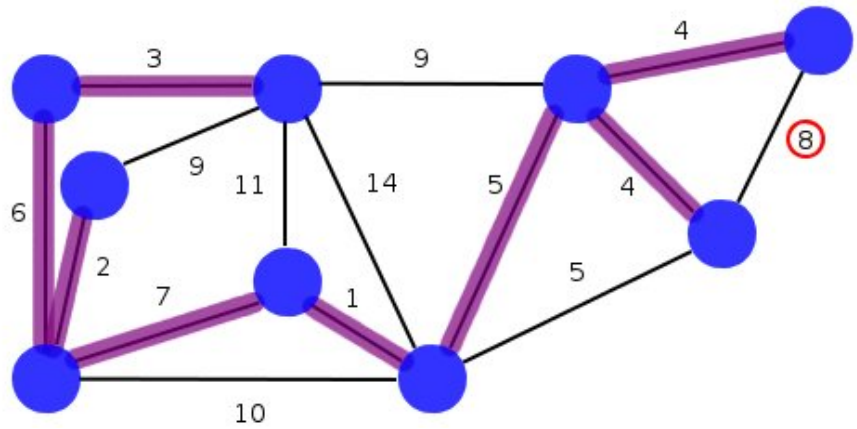
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



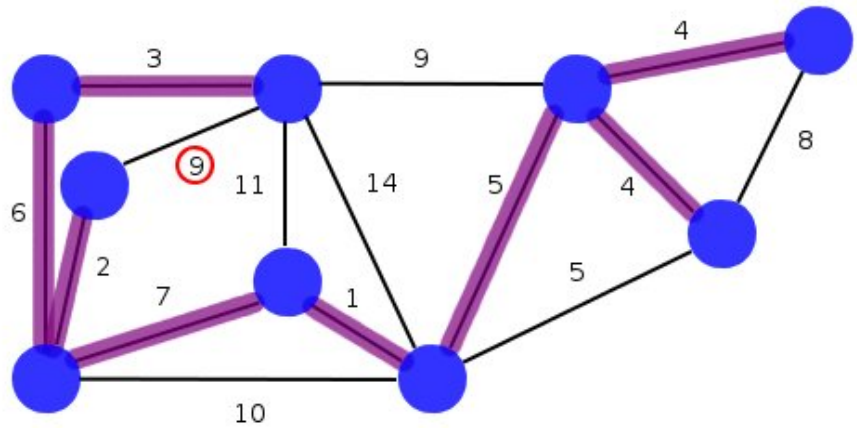
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



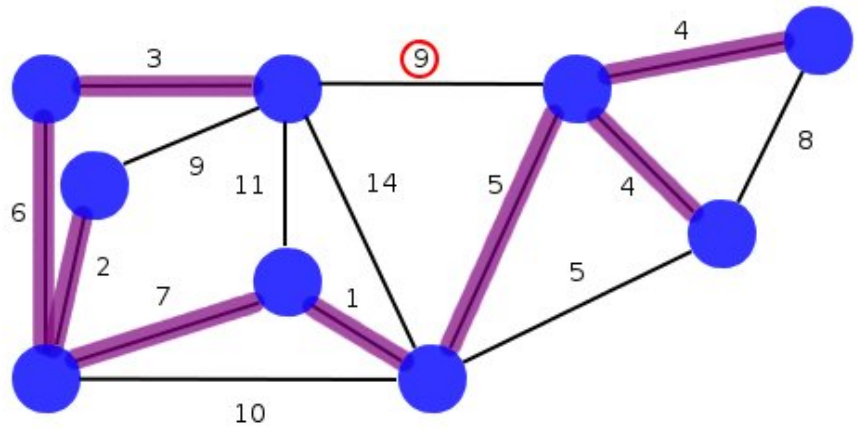
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



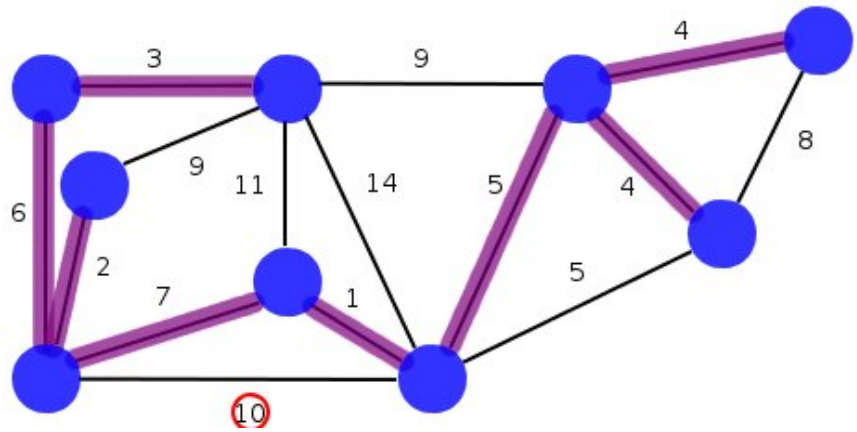
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



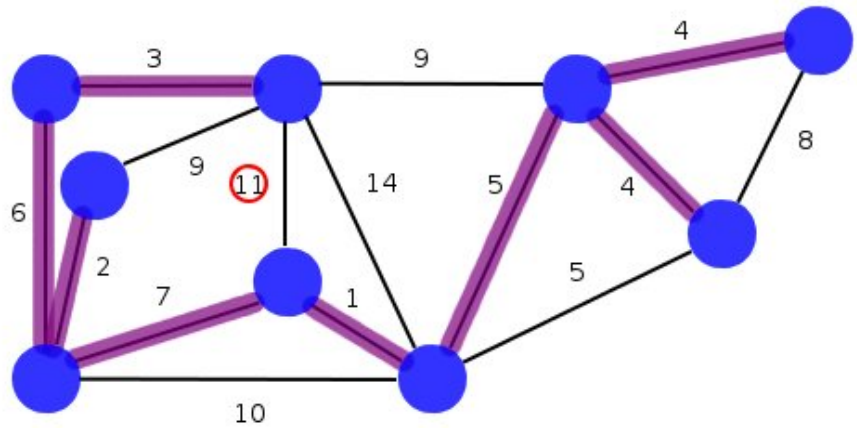
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



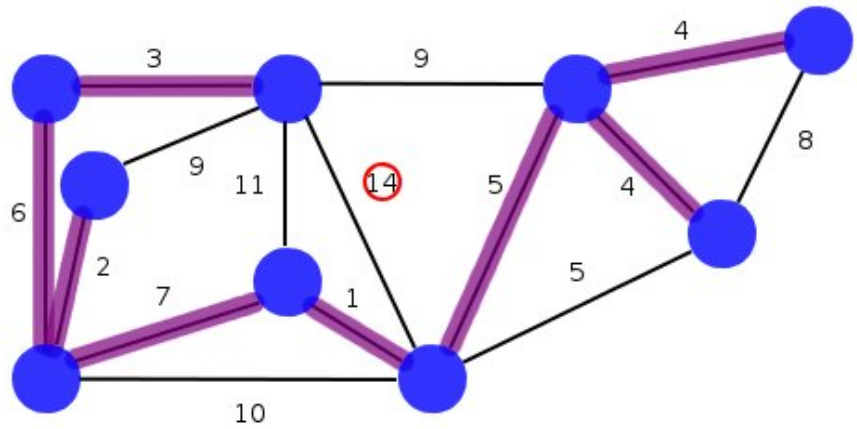
- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Kruskal's Algorithm



- The set of edges is iterated over in weight order
- If the next edge connects two distinct components it is added

Java Implementation

Edges and weighted edges are implemented as explicit classes

```
1   public class Edge {
2       private int from;
3       private int to;
4
5       public Edge(int u, int v) {
6           from = u;
7           to   = v;
8       }
9
10      public int to() {
11          return to;
12      }
13
14      public int from() {
15          return from;
16      }
17  }
```

- A vertex's adjacency list now contains Edge objects
- This allows the weight to be stored in the Edge object

Java Implementation

WeightedEdge is a subclass of Edge

```
1   public class WeightedEdge extends Edge {
2       private int weight;
3
4       public WeightedEdge(int u, int v, int w) {
5           super(u,v);
6           weight = w;
7       }
8
9       public int weight() {
10          return weight;
11      }
12  }
```

- Graph (not shown) has an array of `List<Edge>` objects
- The subclass `WeightedGraph` can add `WeightedEdges` to these lists

Java Implementation

This class makes a new `WeightedGraph` using `g`

```
1  public class KruskalMST extends WeightedGraph {
2
3      public KruskalMST(WeightedGraph g) {
4          super(g.vertices());
5          addEdges(g);
6      }
7
8      private void addEdges(WeightedGraph g) {
9          MinPriorityQueue<WeightedEdge> q = new MinPriorityQueue<WeightedEdge>();
10         ComponentSet          components = new ComponentSet(g.vertices());
11         prioritiseEdges(g, q);
12
13         while(!q.isEmpty()) {
14             WeightedEdge we = q.remove();
15             int u = we.from();
16             int v = we.to();
17             if (!components.sameComponent(u, v)) {
18                 addEdge(u, v, we.weight());
19                 components.connect(u, v);
20             }
21         }
22     }
23
24     ...
25 }
```

Java Implementation

This method adds each edge to the queue

```
1       ...
2
3       private void prioritiseEdges(WeightedGraph g,
4               MinPriorityQueue<WeightedEdge> q) {
5           for (int u = 0; u < vertices(); u++) {
6               for (Edge e : g.adjacent(u)) {
7                   WeightedEdge we = (WeightedEdge) e;
8                   int v = e.to();
9                   if (v > u) { q.add(we.weight(), we); }     // only add once
10              }
11          }
12      }
13  }
```

Java Implementation

This class implements a set of trees

```
1   public class ComponentSet {
2
3       private int[] parent;
4
5       public ComponentSet(int n) {
6           parent = new int[n];
7           Arrays.fill(parent, -1);
8       }
9
10      public boolean sameComponent(int u, int v) {
11          return find(u) == find(v);
12      }
13
14      private int find(int v) {
15          if(parent[v] == -1) { return v; }
16          parent[v] = find(parent[v]);
17          return parent[v];
18      }
19
20      public void connect(int u, int v) {
21          parent[find(u)] = find(v);
22      }
23  }
```

- Initially every vertex has its own set (tree)
- When vertices are connected their trees are merged

Performance of Kruskal's Algorithm

Kruskal's Algorithm executes in $O(E \log_2 V)$ time

- Sorting the edges is $O(E \log_2 E)$
- Set creation, search and union each execute in $O(\log_2 V)$
- So, the tree creation loop is $O(E \log_2 V)$
- Since $|E| \leq |V|^2$, $\log_2 E = O(\log_2 V)$

The set data structure used is referred to as a disjoint set (Cormen) or union-find (Sedgewick) data structure. See books for details.

Prim's Algorithm

Prim's Algorithm always adds edges to the same component

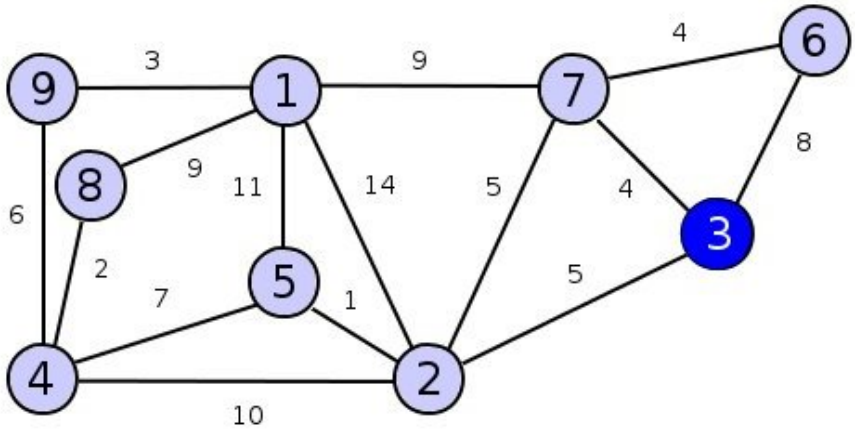
Prim's Algorithm (Input: a connected, weighted graph G and a vertex r)

- $T = (\{r\}, \emptyset)$
- Add all edges (r, v) in G to a queue Q prioritised by min weight
- While T has fewer than $|V| - 1$ edges
  - Remove the next edge (x, y) from Q
  - If y is not in T
    - Add y to T
    - Add (x, y) to T
    - Add all edges (y, v) in G to Q
- Return T and HALT

- It is assumed that edges are ordered
- So, the queue contains edges (x, y) where x must be in T

Prim's Algorithm

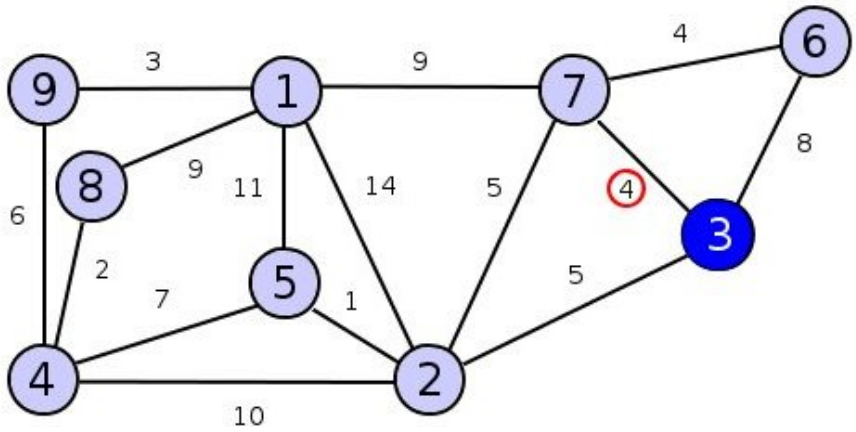
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

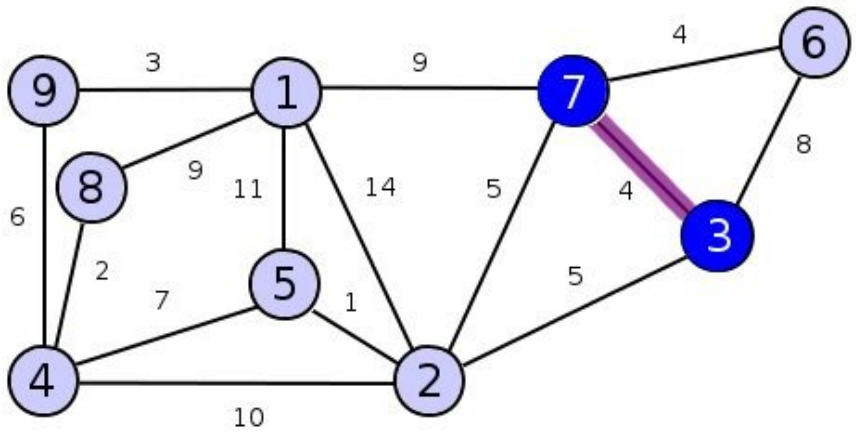
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

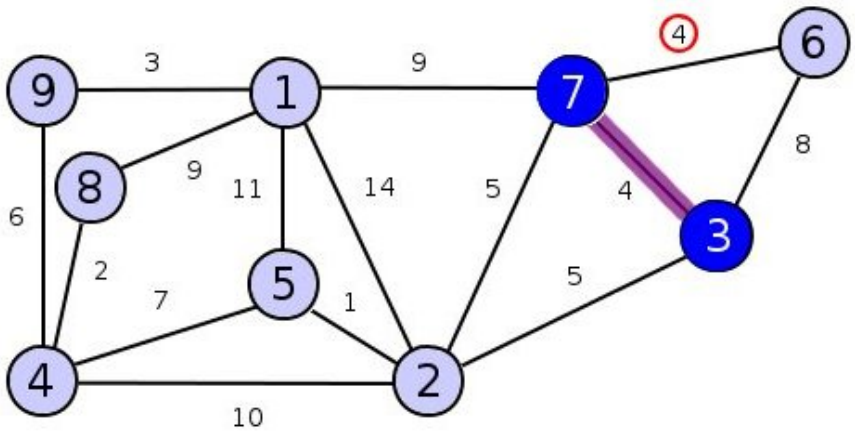
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

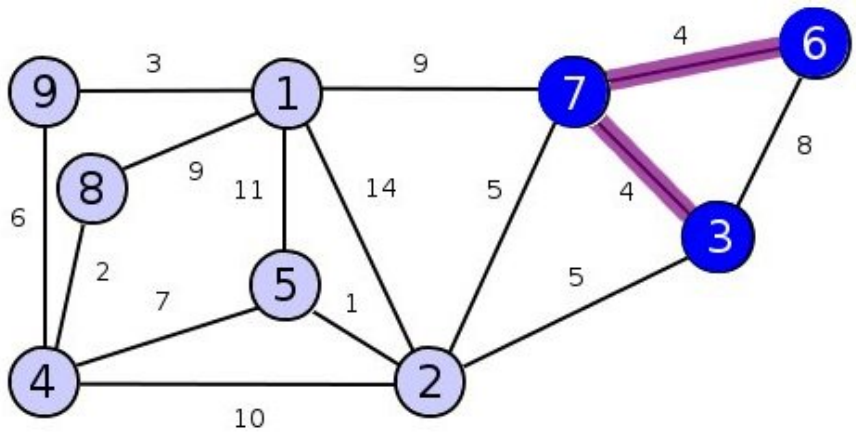
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

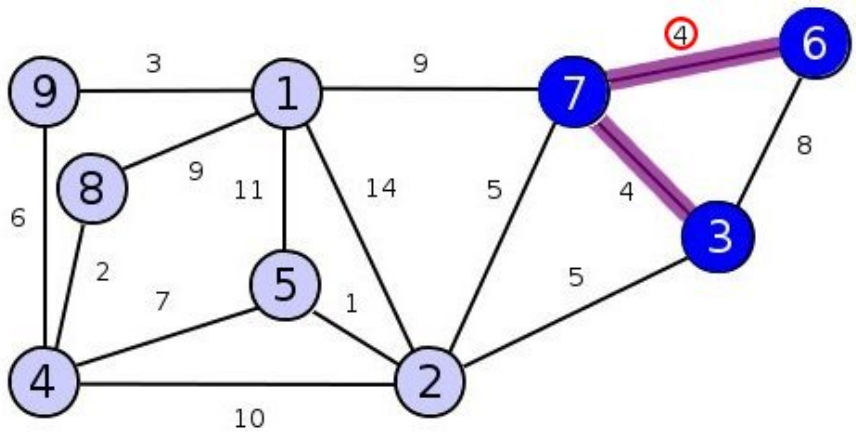
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

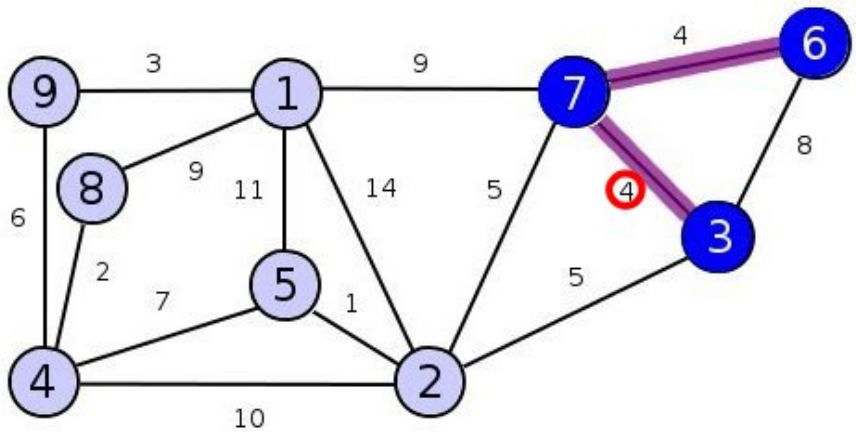
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

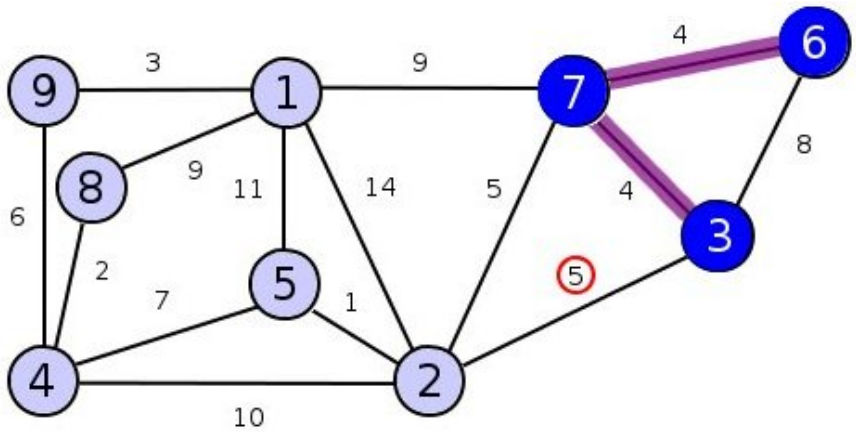
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

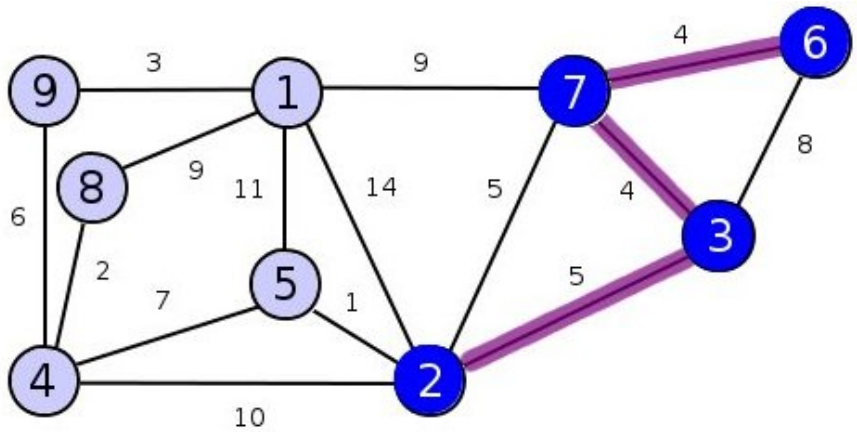
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

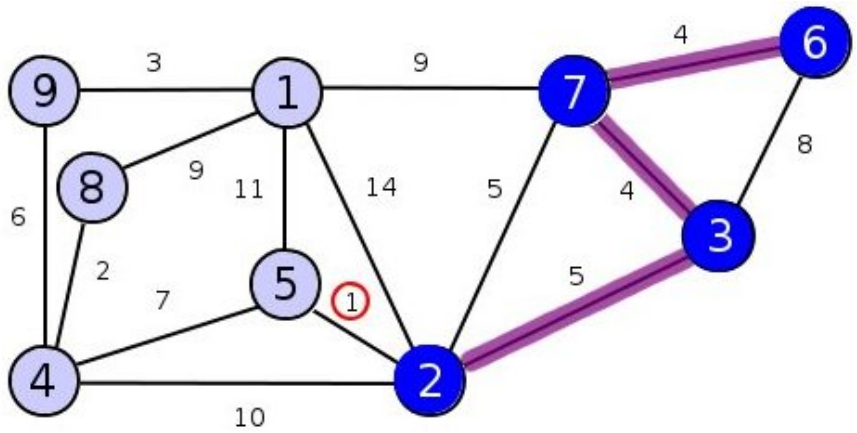
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

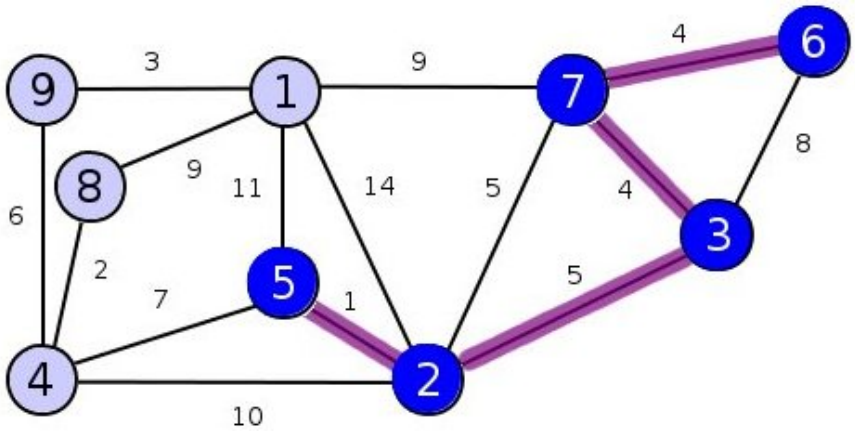
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

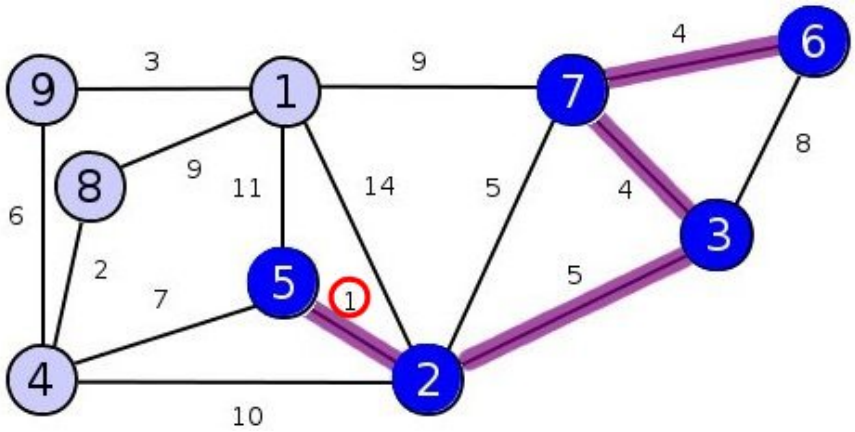
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

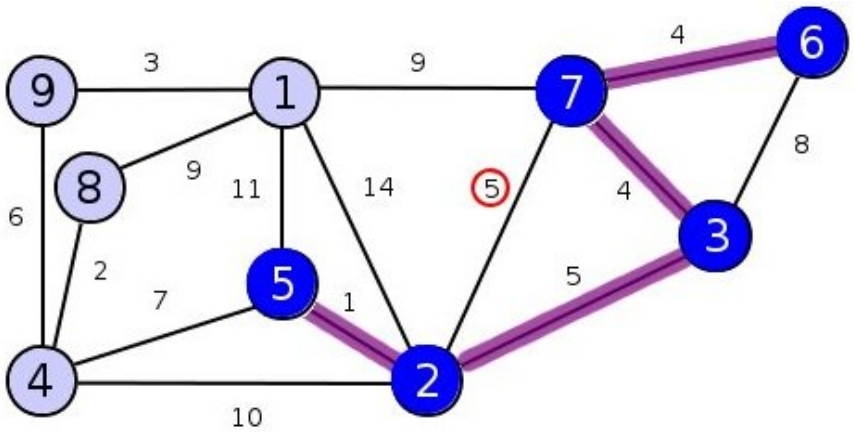
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

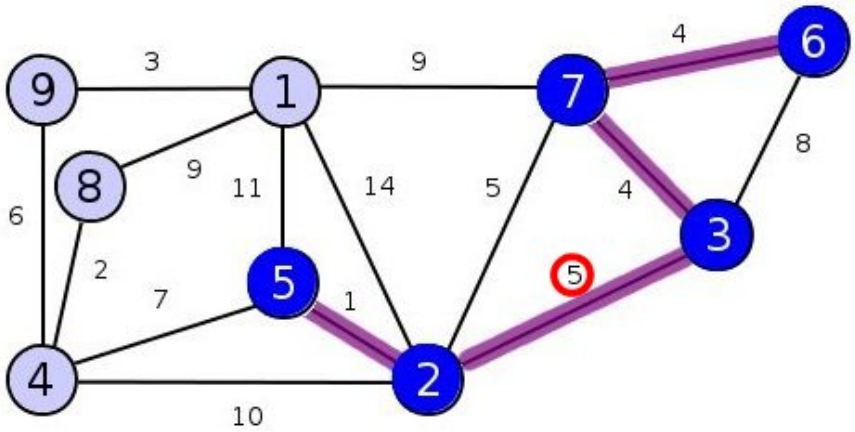
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

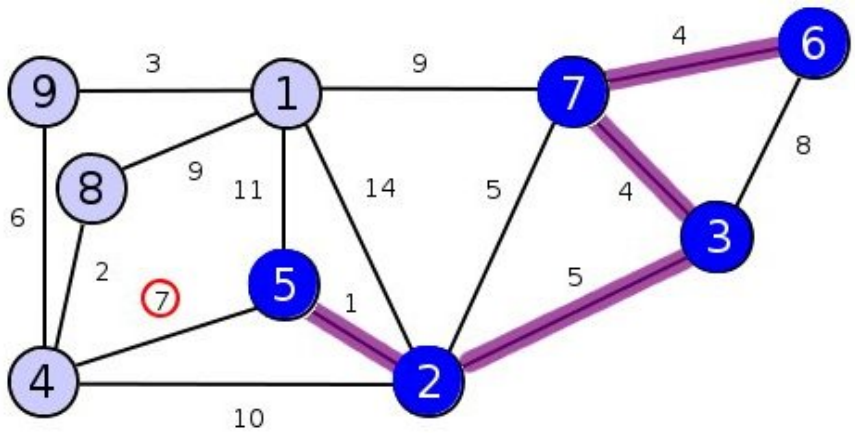
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

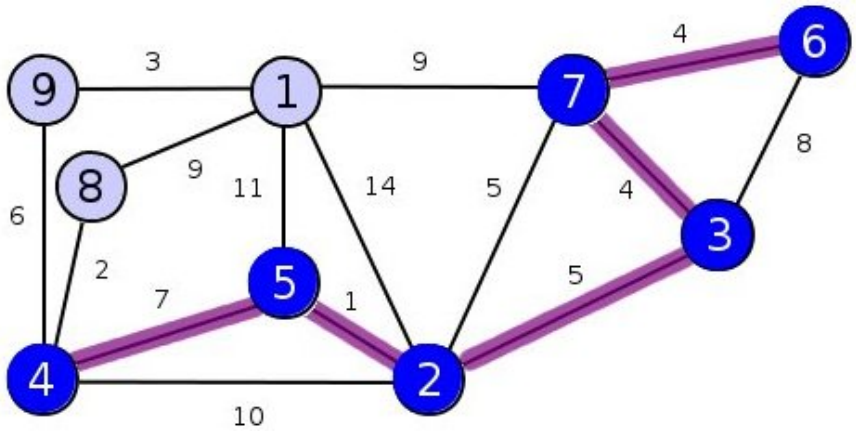
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

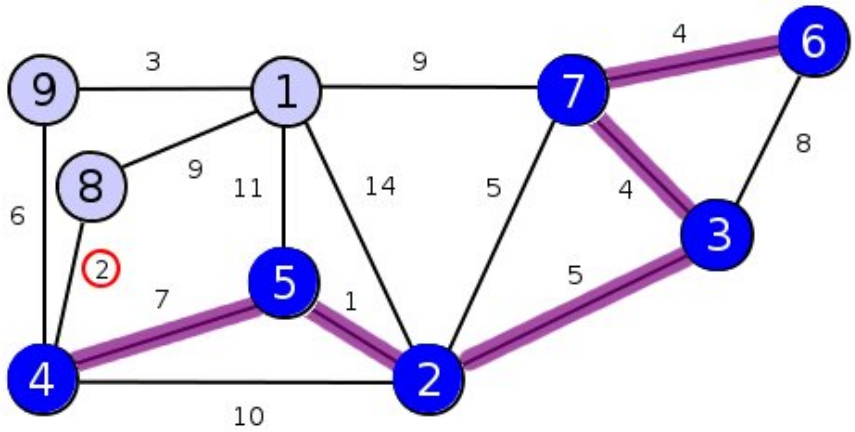
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

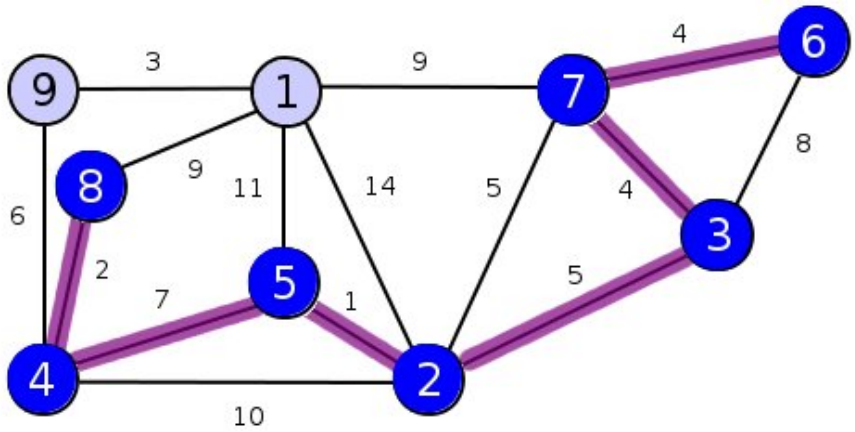
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

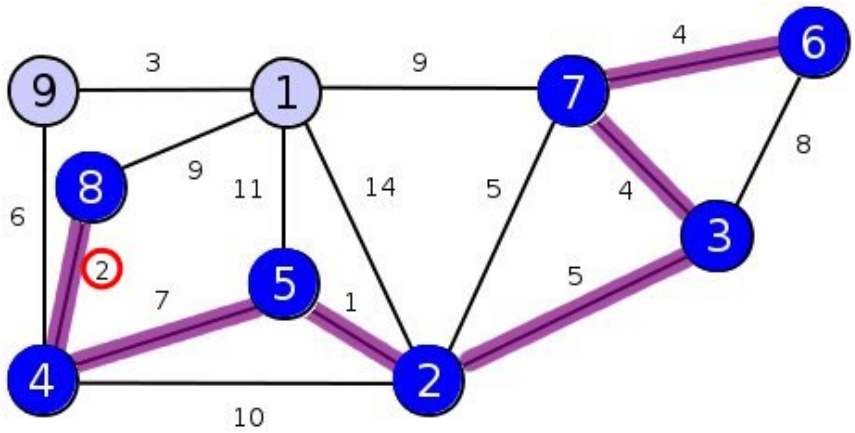
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

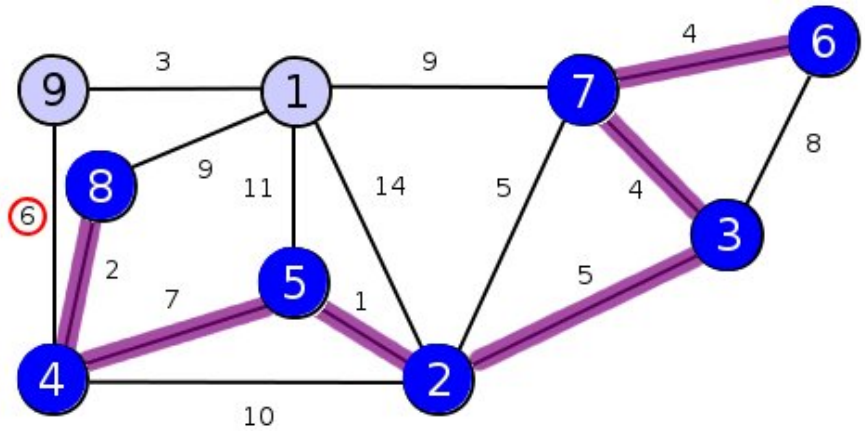
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

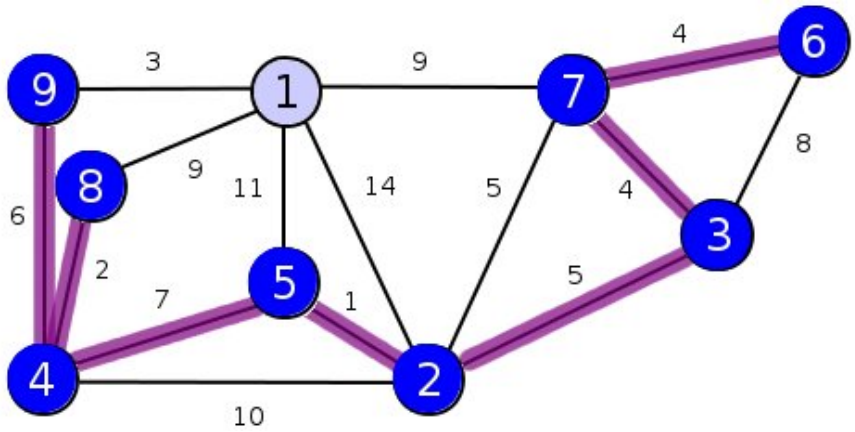
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

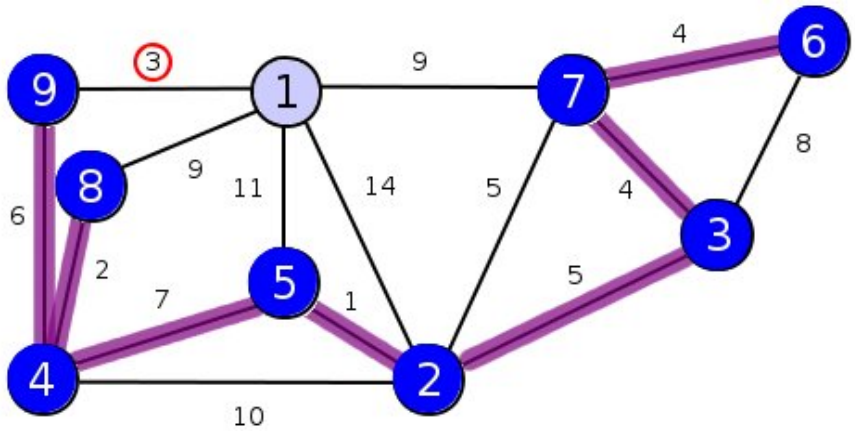
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

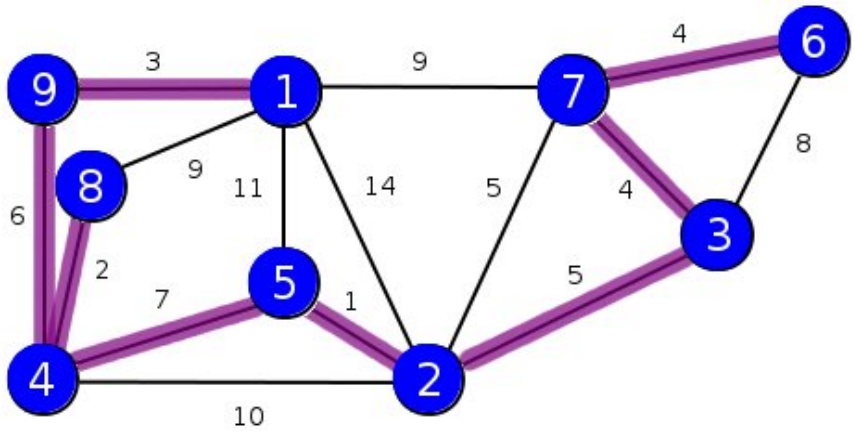
The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Prim's Algorithm

The queue produces the next lowest weight edge incident on T



- If the edge connects T to a new vertex it is a valid greedy choice

Java Implementation

This class only really differs from `KruskalMST` in the `addEdges` method

```
1         ...
2
3         private void addEdges(WeightedGraph g, int r) {
4           MinPriorityQueue<WeightedEdge> q = new MinPriorityQueue<WeightedEdge>();
5           boolean[]                  inTree = new boolean[vertices()];
6
7           inTree[r] = true;
8           collectEdges(r, g, q);         // add edges adjacent to r to the queue
9           int u = r, v = r, size = 1;
10          WeightedEdge next = null;
11
12          while (size++ < vertices()) {
13              while(inTree[v]) {
14                  next = q.remove();
15                  u = next.from();
16                  v = next.to();
17              }
18              addEdge(u, v, next.weight());
19              inTree[v] = true;
20              collectEdges(v, g, q);
21          }
22      }
23      ...
24  }
```

Performance of Prim's Algorithm

Prim's Algorithm also executes in $O(E \log_2 V)$ time assuming a queue implemented as a binary heap

- The queue operations determine the running time
- In the worst case all edges are added to and removed from the queue
- There are faster queue implementations that would improve performance

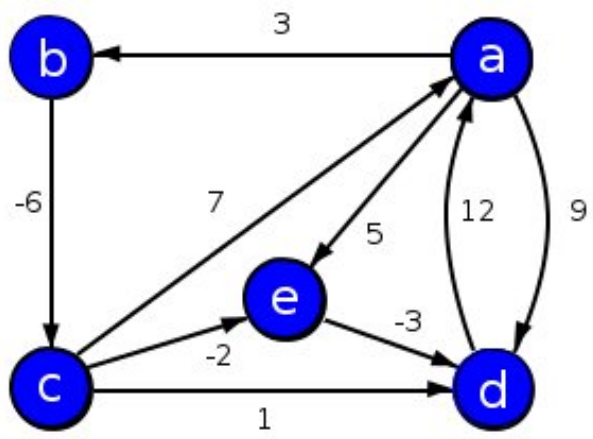
Part IV

Shortest Paths (Again)

More Terminology

Definition (Directed Graph)

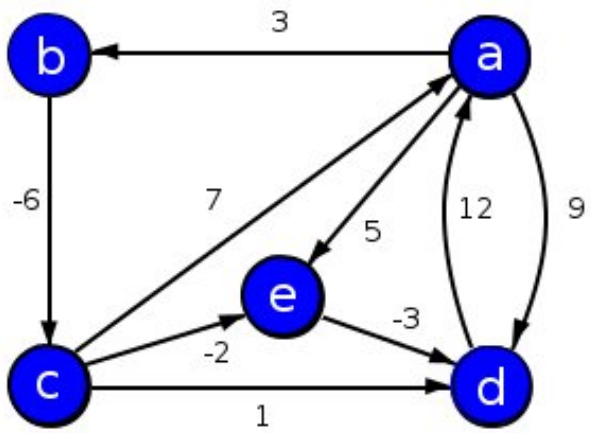
A directed graph is a graph $G = (V, E)$ where V is a set (of objects) and E is a set of ordered pairs of elements of V.



- In a directed graph each edge (u, v) has a direction
- Edges (u, v) and (v, u) can both exist, and have different weights
- An undirected graph can be seen as a special type of directed graph

Shortest Paths

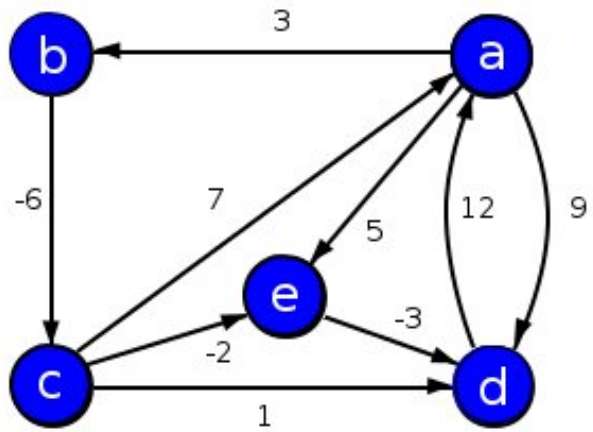
With weighted edges a simple breadth-first search will not find the shortest paths



- The 'shortest' path from a to e is $\langle a, b, c, e \rangle$

Bellman-Ford

The Bellman-Ford algorithm solves the general problem where edges may have negative weights



- A `distance` array is used again
- `distance[v]` is the current estimate of the shortest path to v
- The algorithm proceeds by gradually reducing these estimates

Bellman-Ford

Bellman-Ford (Input: weighted graph G and vertex s)

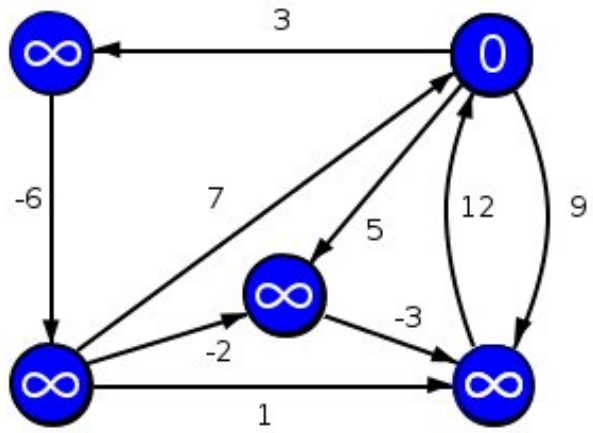
- Set $distance[v] = \infty$ for all vertices
- Set $distance[s] = 0$
- Repeat $|V| - 1$ times:
  - For each edge $e \in E$
    - Relax e
- For each edge $(u, v) \in E$
  - If $distance[v]$ is greater than $distance[u] + w(u, v)$
    - Return *FALSE* and HALT
- Return *TRUE* and HALT

- Relaxing edge (u, v) checks if $s \rightsquigarrow u \rightarrow v$ reduces `distance[v]`
- All edges are relaxed $|V| - 1$ times so all paths are tried
- The algorithm returns *FALSE* if a negative weight cycle occurs

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

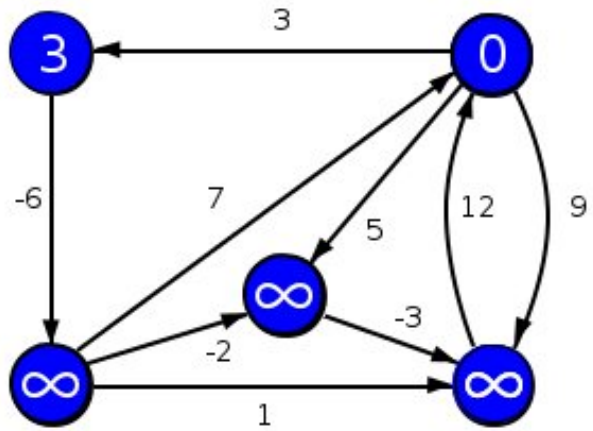


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

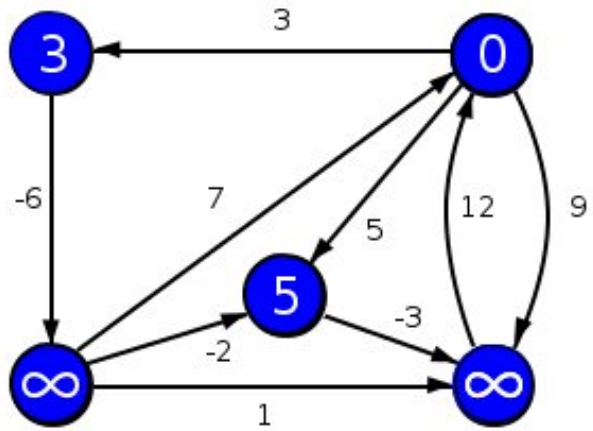


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

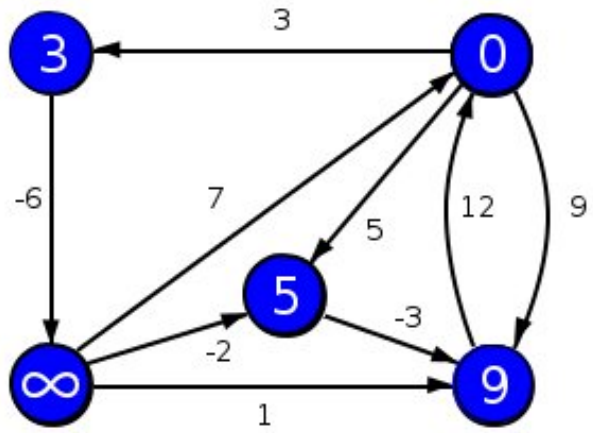


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

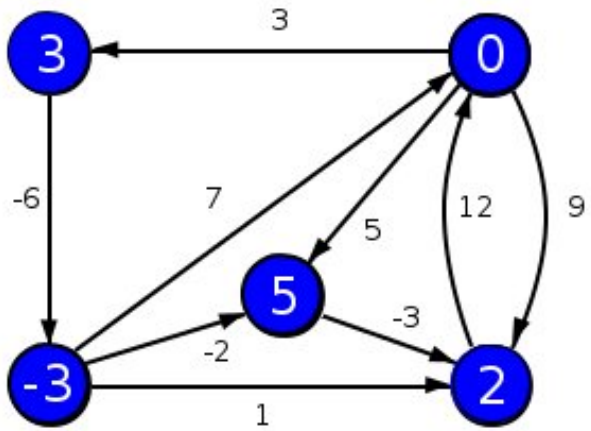


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

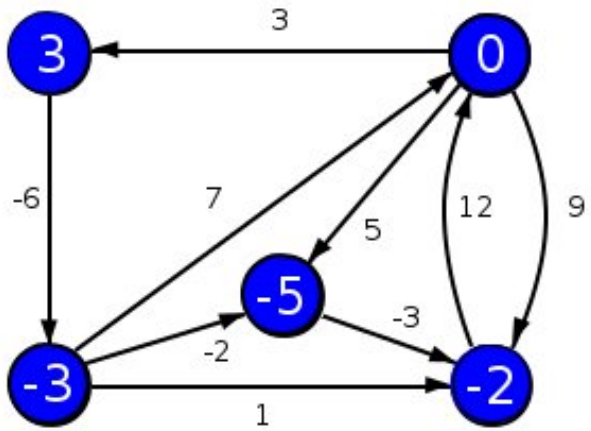


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

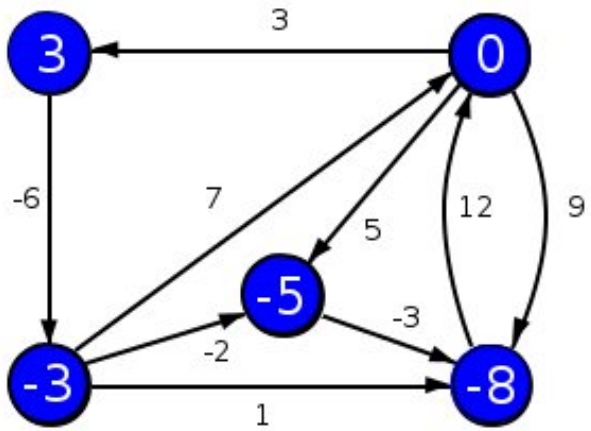


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Relax (Input: weighted edge (u, v))

- If $distance[v]$ is greater than $distance[u] + w(u, v)$ then:
  - $distance[v]$ is $distance[u] + w(u, v)$
  - Parent of v is u

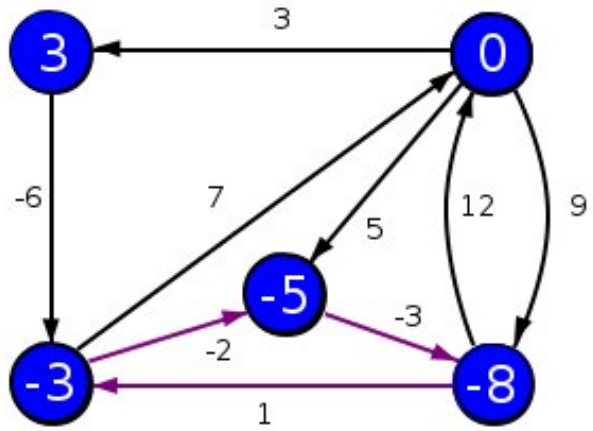


- In iteration i all edges in paths containing i vertices have been relaxed
- The maximum vertices in any path is $|V| - 1$

Bellman-Ford

Definition (Negative Weight Cycle)

A path $C = \langle v_1, v_2, \ldots, v_n \rangle$ in a directed graph is a negative weight cycle iff C is a cycle and $\sum_{i=1}^{n-1} w(v_i, v_{i+1}) < 0$.



If a directed graph G contains a negative weight cycle $\langle v_1, v_2, \ldots, v_n \rangle$ then:

- The shortest paths to all vertices reachable from $v_1, \ldots, v_n$ are undefined
- In this case Bellman-Ford will return *FALSE*

Java Implementation

The `WeightedGraph` class is updated to represent a directed graph

```
1   public class WeightedGraph extends Graph{
2
3       ...
4
5       public void addEdge(int u, int v, int w) {
6           if (!isVertex(u) || ! isVertex(v)) {
7               throw new IndexOutOfBoundsException();
8           }
9           addEdgeTo(u, new WeightedEdge(u, v, w));
10      }
11  }
```

- Adding an edge now only adds it in one direction
- `UndirectedWeightedGraph` extends this class and overrides this method

Java Implementation

This abstract base class initialises and calls `findShortestPaths()`

```
1  public abstract class SingleSourceShortestPaths {
2
3      protected final static int MAX = Integer.MAX_VALUE;
4
5      private int             source;
6      protected WeightedGraph graph;
7      private int[]           parent;
8      private int[]           distance;
9
10     public SingleSourceShortestPaths(WeightedGraph g, int s) {
11         graph    = g;
12         source   = s;
13         parent   = new int[g.vertices()];
14         distance = new int[g.vertices()];
15         Arrays.fill(parent, -1);
16         Arrays.fill(distance, MAX);
17         distance[s] = 0;
18         findShortestPaths();
19     }
20
21     ...
```

- The `MAX_VALUE` int is used to represent infinity

Java Implementation

The base class also implements the `relax()` method

```
1     ...
2
3     protected abstract void findShortestPaths();
4
5     protected void relax(WeightedEdge we) {
6         int u = we.from();
7         int v = we.to();
8         int distToVViaU = plus(distance[u], we.weight());
9         if (distance[v] > distToVViaU) {
10            distance[v] = distToVViaU;
11            parent[v]   = u;
12        }
13    }
14
15    ...
```

- `plus()` is used to handle the case when a path is infinite
- `findShortestPaths()` will be implemented by concrete subclasses

Java Implementation

`relax()` and `plus()` are both `protected` so they can be called by subclasses

```
1     ...
2
3     protected int plus(int a, int b) {
4         if (a == MAX || b == MAX) { return MAX; }
5         return a + b;
6     }
7
8     public int distanceTo(int v) {
9         if (!graph.isVertex(v)) { throw new IllegalArgumentException(); }
10        return distance[v];
11    }
12 }
```

Java Implementation

`BellmanFordShortestPaths` implements `findShortestPaths` and also checks for negative weight cycles

```
1  public class BellmanFordShortestPaths extends SingleSourceShortestPaths {
2
3      private boolean noPaths;
4
5      public BellmanFordShortestPaths(WeightedGraph g, int s) {
6          super(g,s);
7          checkForNegativeWeightCycles();
8      }
9
10     @Override
11     protected void findShortestPaths() {
12         int V = graph.vertices();
13         for (int i = 1; i < V; i++) {
14             for (int u = 0; u < V; u++) {
15                 for (Edge we : graph.adjacent(u)) {
16                     relax((WeightedEdge) we);
17                 }
18             }
19         }
20     }
21
22     ...
```

Java Implementation

The `noPaths` flag is set if any edge fails the negative cycle test

```
1     ...
2
3     private void checkForNegativeWeightCycles() {
4         int V = graph.vertices();
5         for (int u = 0; u < V; u++) {
6             for (Edge we : graph.adjacent(u)) {
7                 if (invalidPath((WeightedEdge) we)) {
8                     noPaths = true;
9                     return;
10                }
11            }
12        }
13    }
14
15    private boolean invalidPath(WeightedEdge we) {
16        int u = we.from();
17        int v = we.to();
18        int distToVViaU = plus(distanceTo(u), we.weight());
19        return distanceTo(v) > distToVViaU;
20    }
21 }
```

Java Implementation

The Bellman-Ford class overrides the public `distanceTo` method

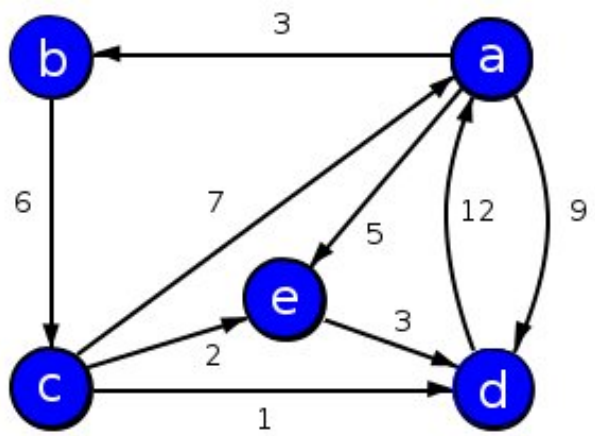
```
1     ...
2
3     public boolean hasNegativeWeightCycle() {
4         return noPaths;
5     }
6
7     public int distanceTo(int v) {
8         if (noPaths) { throw new UndefinedShortestPathsException(); }
9         return super.distanceTo(v);
10    }
11
12    ...
```

- A custom exception class is used

Dijkstra's Algorithm

If G has non-negative edges only then we can use Dijkstra's Algorithm

- Bellman-Ford relaxes every edge of every path
- The running time of Bellman-Ford is $O(VE)$
- Dijkstra's algorithm instead uses a greedy strategy



Dijkstra's Algorithm

Dijkstra's algorithm maintains a set of vertices whose $distance[v]$ is correct

Dijkstra (Input: weighted graph G, vertex s)

- $distance[v] = \infty$ for all vertices
- $distance[s] = 0$
- $S = \emptyset$
- While $V - S \neq \emptyset$
  - u is the vertex in $V - S$ with least $distance[u]$
  - For each vertex v adjacent to u
    - Relax (u, v)
  - $S = S \cup \{u\}$
- HALT

- The next vertex added to S is the one with the least $distance[u]$
- This value will not be reduced further, so is it correct?

Correctness

In the following, the function p represents the (actual) length of the shortest path from the source to a given vertex

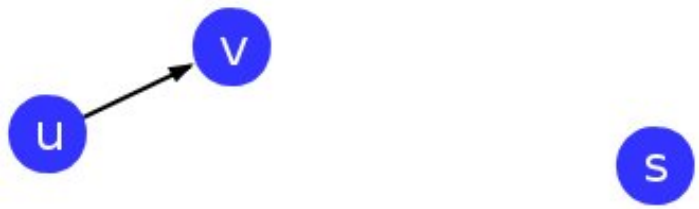
- If there is no path $s \rightsquigarrow v$, then $p(v) = \infty$
- $\infty + x = \infty$, for all $x \in \mathbb{R}$

Theorem (Correctness of Dijkstra)

At the start of the while loop of Dijkstra's algorithm, run on weighted, directed graph $G = (V, E)$ with non-negative weight function w, and vertex $s \in V$: if distance$[v] = p(v)$ for all vertices $v \in S$, then distance$[u] = p(u)$ for u, the next vertex added to S.

Proof

First we prove two useful properties



Lemma (Triangle Lemma)

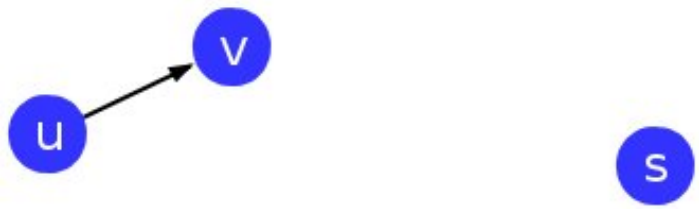
Let $G = (V, E)$ be a weighted, directed graph with weight function w, and source vertex s. If (u, v) is an edge in E, then $p(v) \leq p(u) + w(u, v)$.

Proof.

If there is no path $s \rightsquigarrow u$, then $p(u) = \infty$, so $p(v) \leq p(u)$ and the lemma holds. If there is a path $s \rightsquigarrow u$, then $s \rightsquigarrow u \rightarrow v$ is a path to v. The length of one such path to v is $p(u) + w(u, v)$. The *shortest* path to v cannot be *longer* than this, so the lemma also holds in this case. □

Proof

First we prove two useful properties



Lemma (Triangle Lemma)

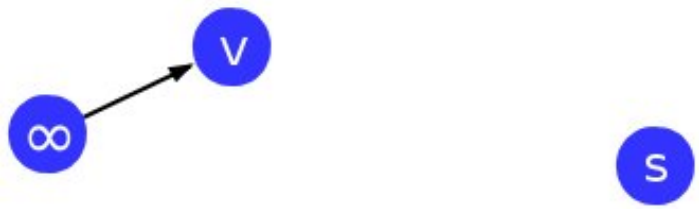
Let $G = (V, E)$ be a weighted, directed graph with weight function w, and source vertex s. If (u, v) is an edge in E, then $p(v) \leq p(u) + w(u, v)$.

Proof.

If there is no path $s \rightsquigarrow u$, then $p(u) = \infty$, so $p(v) \leq p(u)$ and the lemma holds. If there is a path $s \rightsquigarrow u$, then $s \rightsquigarrow u \rightarrow v$ is a path to v. The length of one such path to v is $p(u) + w(u, v)$. The *shortest* path to v cannot be *longer* than this, so the lemma also holds in this case. □

Proof

First we prove two useful properties



Lemma (Triangle Lemma)

Let $G = (V, E)$ be a weighted, directed graph with weight function w, and source vertex s. If (u, v) is an edge in E, then $p(v) \leq p(u) + w(u, v)$.

Proof.

If there is no path $s \rightsquigarrow u$, then $p(u) = \infty$, so $p(v) \leq p(u)$ and the lemma holds. If there is a path $s \rightsquigarrow u$, then $s \rightsquigarrow u \rightarrow v$ is a path to v. The length of one such path to v is $p(u) + w(u, v)$. The *shortest* path to v cannot be *longer* than this, so the lemma also holds in this case. □

Proof

First we prove two useful properties

Lemma (Triangle Lemma)

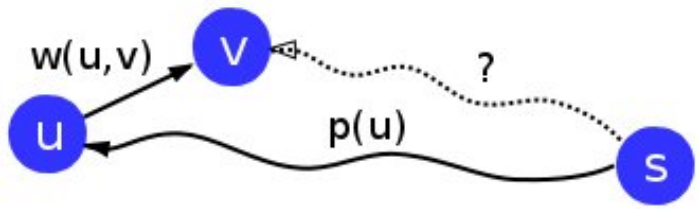
Let $G = (V, E)$ be a weighted, directed graph with weight function w, and source vertex s. If (u, v) is an edge in E, then $p(v) \leq p(u) + w(u, v)$.

Proof.

If there is no path $s \rightsquigarrow u$, then $p(u) = \infty$, so $p(v) \leq p(u)$ and the lemma holds. If there is a path $s \rightsquigarrow u$, then $s \rightsquigarrow u \rightarrow v$ is a path to v. The length of one such path to v is $p(u) + w(u, v)$. The *shortest* path to v cannot be *longer* than this, so the lemma also holds in this case. □

Proof

First we prove two useful properties



Lemma (Triangle Lemma)

Let $G = (V, E)$ be a weighted, directed graph with weight function w, and source vertex s. If (u, v) is an edge in E, then $p(v) \leq p(u) + w(u, v)$.

Proof.

If there is no path $s \rightsquigarrow u$, then $p(u) = \infty$, so $p(v) \leq p(u)$ and the lemma holds. If there is a path $s \rightsquigarrow u$, then $s \rightsquigarrow u \rightarrow v$ is a path to v. The length of one such path to v is $p(u) + w(u, v)$. The *shortest* path to v cannot be *longer* than this, so the lemma also holds in this case. □

Proof

This lemma shows that $distance[u]$ is always an upper bound for $p(u)$

Lemma (Upper Bound Lemma)

Let $G = (V, E)$ be a weighted, directed graph with weight function w, and source vertex s. If $distance[s]$ is initialised to 0 and $distance[v]$, for all $v \in V$ where $v \neq s$, is initialised to ∞, then $distance[u] \geq p(u)$, for all $u \in V$, after relaxing any sequence of edges in G.

Proof.

Firstly, consider a sequence of 0 relaxed edges.

- $distance[u] = \infty$, for $u \neq s$
- $distance[s] = 0$

If s is part of a negative weight cycle, then $p(s) = -\infty$, otherwise $p(s) = 0$. So, $distance[u] \geq p(u)$ for all $u \in V$ in this case. □

Proof

Proof (continued).

Now consider the relaxation of edge (x, y) within some sequence of relaxations.

- Assume $distance[u] \geq p(u)$ for all $u \in V$, prior to relaxing (x, y)

When (x, y) is relaxed either all $distance[u]$ are unchanged, or $distance[y] = distance[x] + w(x, y)$. In the latter case:

- $distance[y] = distance[x] + w(x, y)$, so
- $distance[y] \geq p(x) + w(x, y)$, by the assumption, and
- $distance[y] \geq p(y)$, by the Triangle Lemma

So after relaxing (x, y), $distance[u] \geq p(u)$ still holds for all vertices in G, and by the principle of induction $distance[u] \geq p(u)$ is always true for any sequence of edge relaxations. □

Proof

Theorem (Correctness of Dijkstra)

At the start of the while loop of Dijkstra's algorithm, run on weighted, directed graph $G = (V, E)$ with non-negative weight function w, and vertex $s \in V$: if $distance[v] = p(v)$ for all vertices $v \in S$, then $distance[u] = p(u)$ for u, the next vertex added to S.

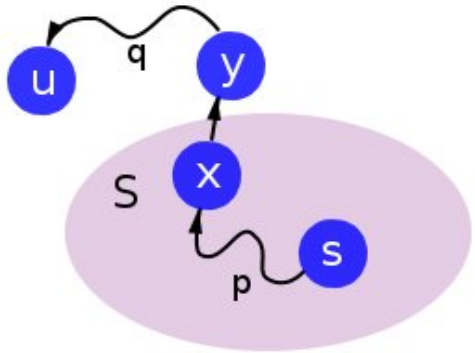
Proof.

If there is no path $s \rightsquigarrow u$ then $p(u) = \infty$. Since:

- $distance[u] \geq p(u)$, by the Upper Bound Lemma, then
- $distance[u] = \infty$, so
- $distance[u] = p(u)$.

and the theorem is true. □

Proof



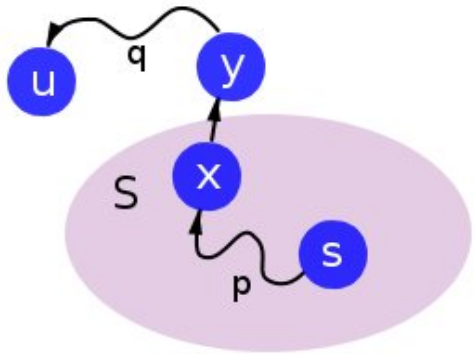
Proof (continued).

If there is a path $s \rightsquigarrow u$, then consider the shortest such path. Let this path be $s \rightsquigarrow^p x \rightarrow y \rightsquigarrow^q u$, where y is the first vertex on the path not in S. First, it is shown that $distance[y] = p(y)$, as follows. $s \rightsquigarrow^p x \rightarrow y$ must be a shortest path from s to y. (Or there would be a shorter path to u.) Then,

- $distance[x] = p(x)$
- $distance[y] = distance[x] + w(x, y) = p(x) + w(x, y)$

since x is in S and (x, y) was relaxed when x was added to S. □

Proof



Proof (continued).

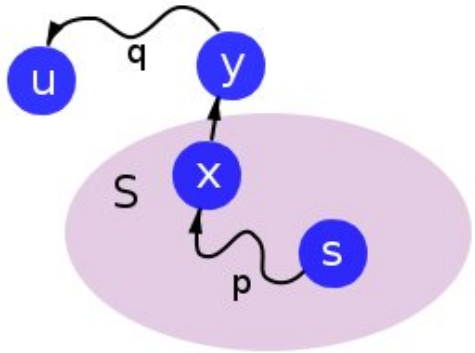
And, since $s \rightsquigarrow^{p} x \rightarrow y$ is a shortest path from s to y, then:

- $p(y) = p(x) + w(x, y) = distance[y]$

Next we show that $distance[u] = distance[y] = p(y) = p(u)$ using the observations that

- (1) $distance[u] \leq distance[y]$, since u is added next to S
- (2) $p(y) \leq p(u)$, since all edges are non-negative.

Proof



Proof (continued).

So, beginning with Observation (1):

- $distance[u] \leq distance[y]$, and therefore
- $distance[u] \leq p(y)$, and
- $distance[u] \leq p(u)$, by Observation (2).

But $distance[u] \geq p(u)$ by the Upper Bound Lemma, so $distance[u] = p(u)$ and the theorem is true. □

Implementation

As before, this class extends the abstract base type

```
1  public class DijkstraShortestPaths extends SingleSourceShortestPaths {
2
3      private QueueElem[] elems;
4
5      public DijkstraShortestPaths(WeightedGraph g, int s) {
6          super(g, s);
7      }
8
9      ...
```

- The base class initialises the `distance` array
- Vertices are placed into a queue, prioritised by `distance[v]`
- The `elems` array stores references to the queue elements

Implementation

This method initialises the queue

```
1       ...
2
3       private MinPriorityQueue<Integer> initQueue() {
4           MinPriorityQueue<Integer> q = new MinPriorityQueue<Integer>();
5           for (int v = 0; v < graph.vertices(); v++) {
6               elems[v] = q.add(distanceTo(v), v);
7           }
8           return q;
9       }
10
11  }
```

- Each vertex is added to the queue, with its distance as key
- The queue class was modified to return a reference to the new element
- These references are stored in `elems`, by vertex value

Implementation

```
1       ...
2
3       protected void findShortestPaths() {
4           elems = new QueueElem[graph.vertices()];
5           MinPriorityQueue<Integer> q = initQueue();
6
7           int u, v;
8           while(!q.isEmpty()) {
9               u = q.remove();
10              elems[u] = null;
11              for (Edge e: graph.adjacent(u)) {
12                  relax((WeightedEdge) e);
13                  v = e.to();
14                  if (elems[v] != null) { q.decreaseKey(elems[v], distanceTo(v)); }
15              }
16          }
17      }
18
19      ...
```

- `initQueue()` is called after creating `elems`
- As each vertex is removed from the queue adjacent edges are relaxed
- The keys of vertices still in the queue are updated as edges are relaxed

Performance

The running time of Dijkstra's algorithm depends on the way in which the ordering of the vertices is managed

- There is one iteration through the graph vertices
- Each edge is relaxed once, giving an aggregate of $|E|$

With a binary-heap-based priority queue adding, removing and updating (changing key) all run in $O(\log_2 V)$ time.

- Overall running time is then $O(E \log_2 V)$