BFS

Breadth First Search

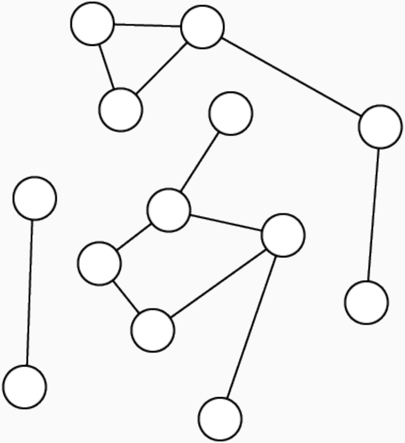
Daniel Graf (Slides by Benjamin Schmid)

2017-11-04

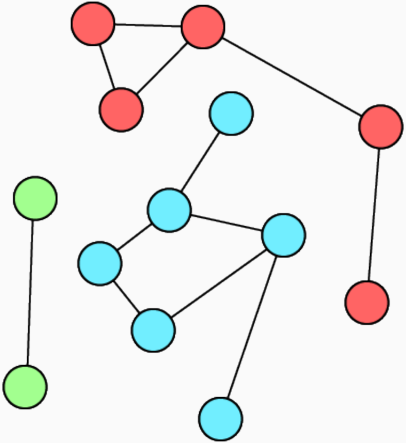
Swiss Olympiad in Informatics

Introduction

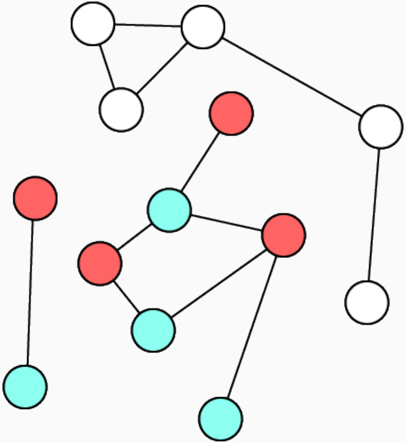
Graph traversal



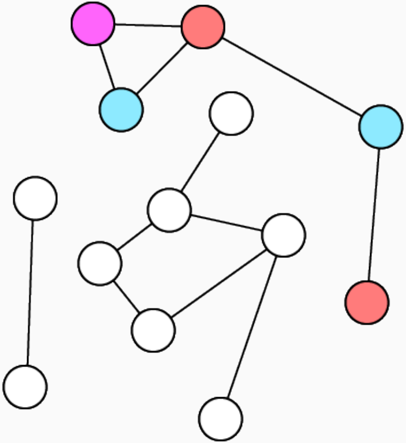
Components



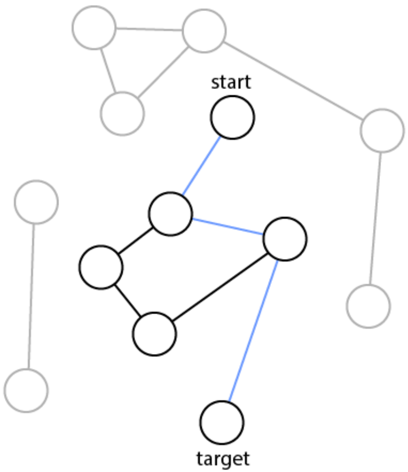
Bipartite



Bipartite

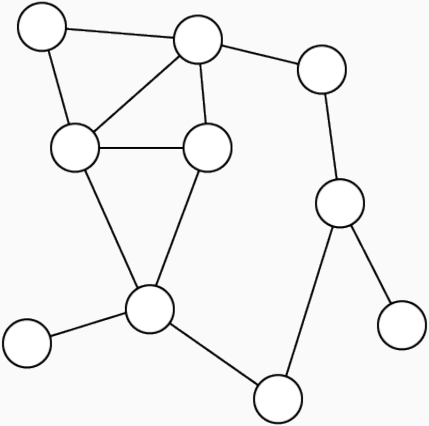


Shortest path

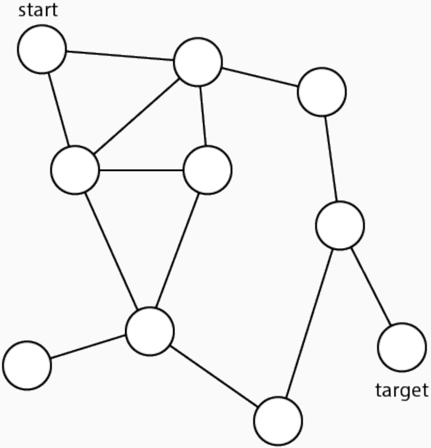


Shortest path

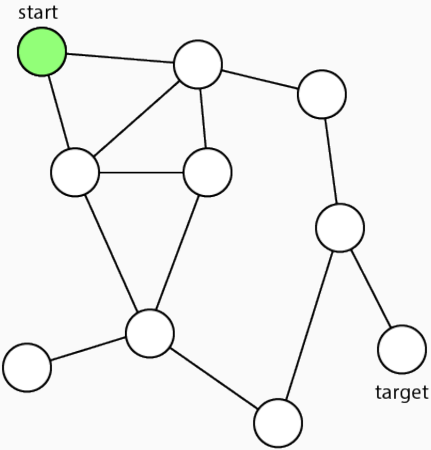
Shortest path



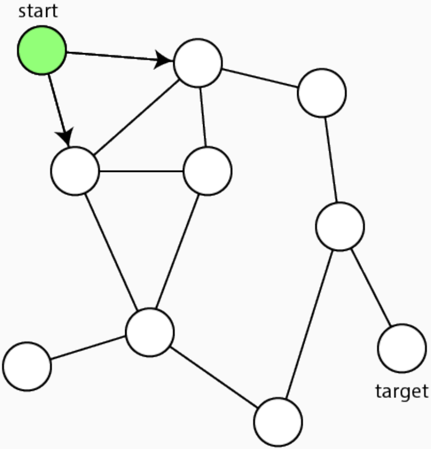
Shortest path



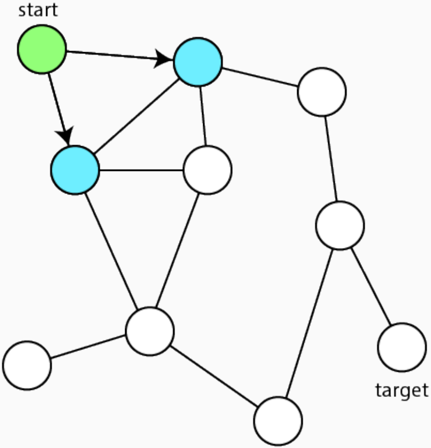
Shortest path



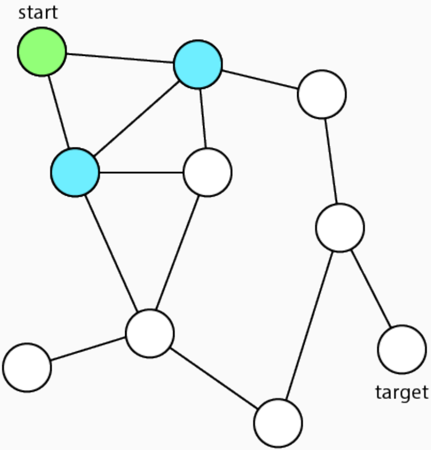
Shortest path



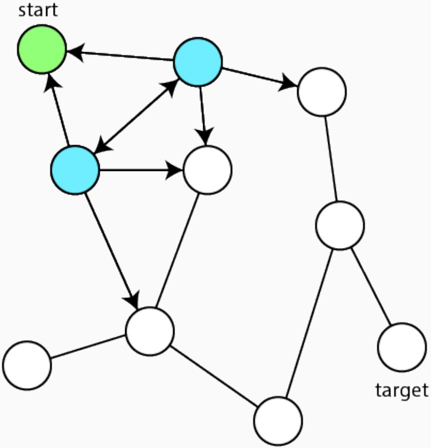
Shortest path



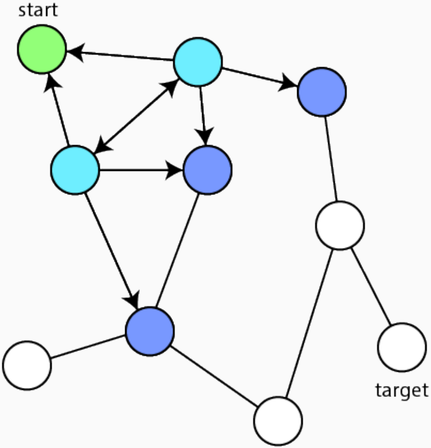
Shortest path



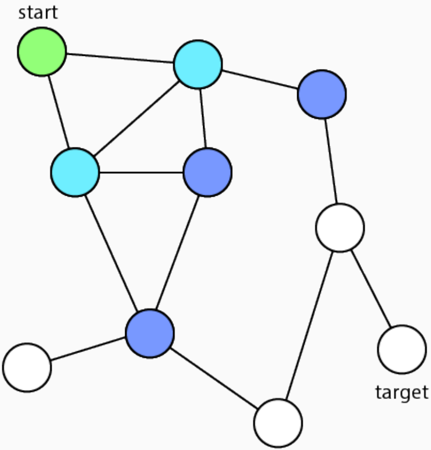
Shortest path



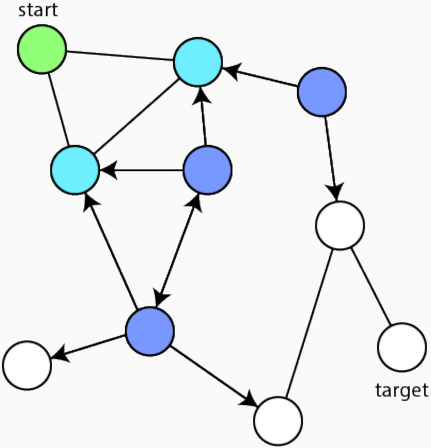
Shortest path



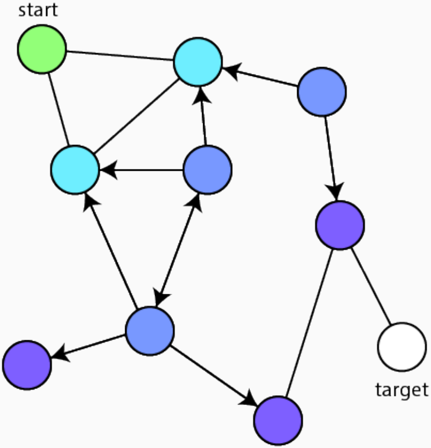
Shortest path



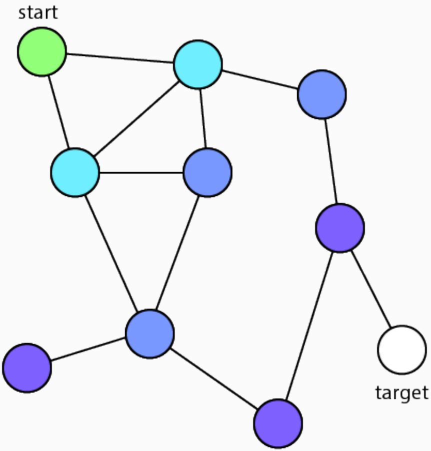
Shortest path



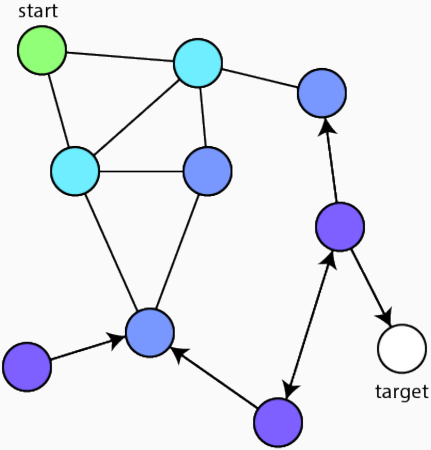
Shortest path



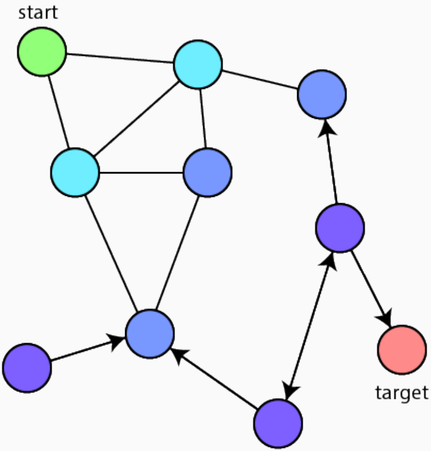
Shortest path



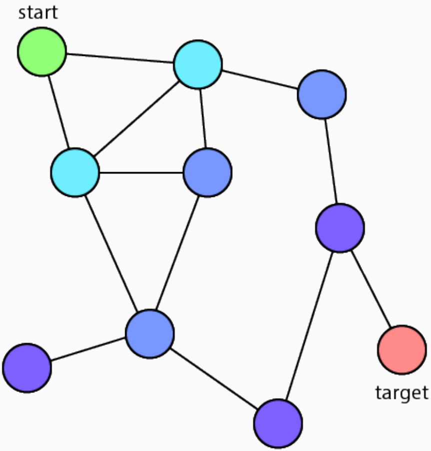
Shortest path



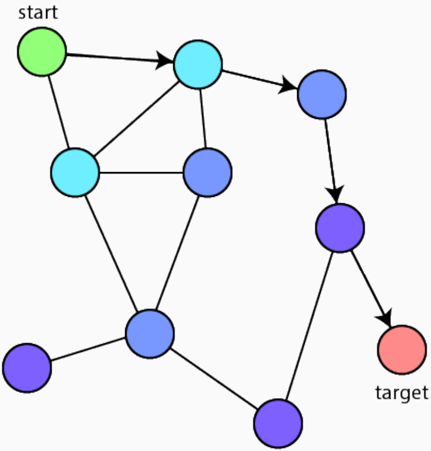
Shortest path



Shortest path



Shortest path

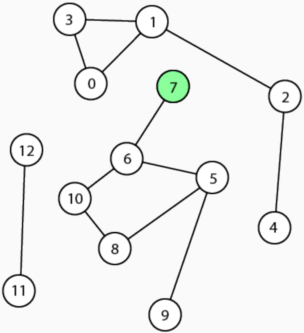


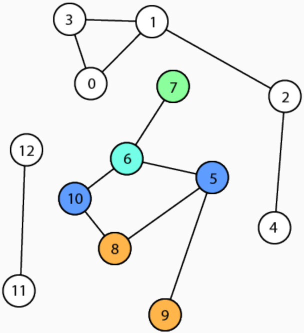
BFS

- Breadth First Search (BFS)
- First explore breadth
- Nodes visited in order of distance to start
- Thus find shortest path (if equal lengths)

BFS Implementation

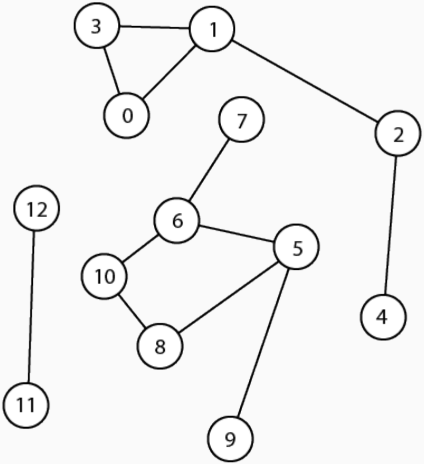
Find Component





Generations / Queue

Generations:
[ ]

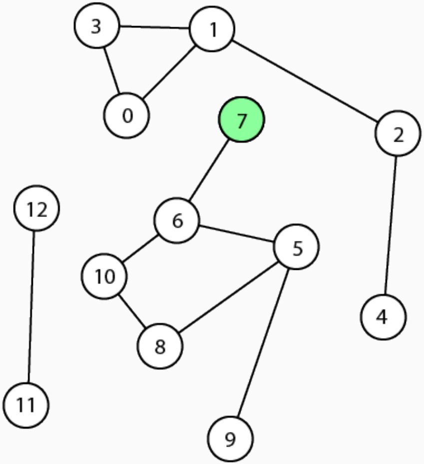


Generations / Queue

Generations:

[7]

[ ]

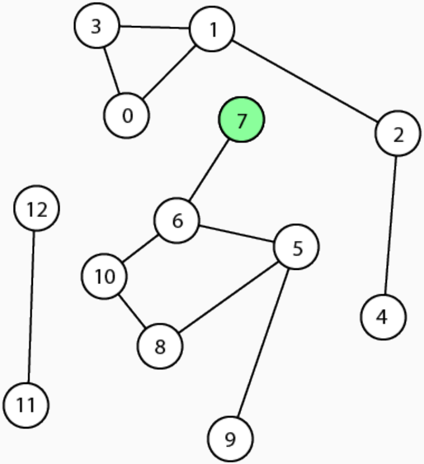


Generations / Queue

Generations:

[7]

[6]

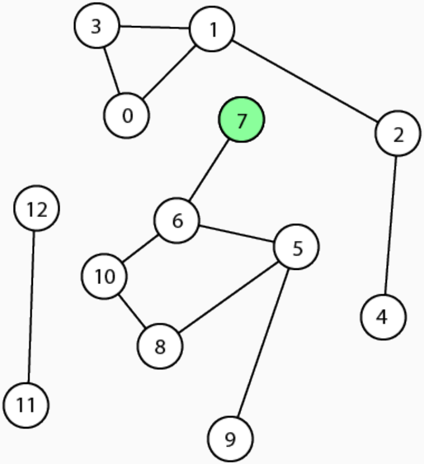


Generations / Queue

Generations:

[7]

[6]



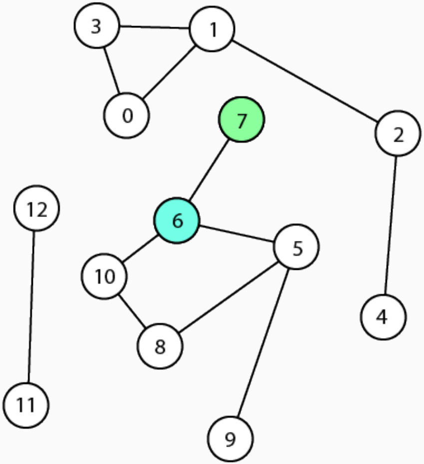
Generations / Queue

Generations:

[7]

[6]

[ ]



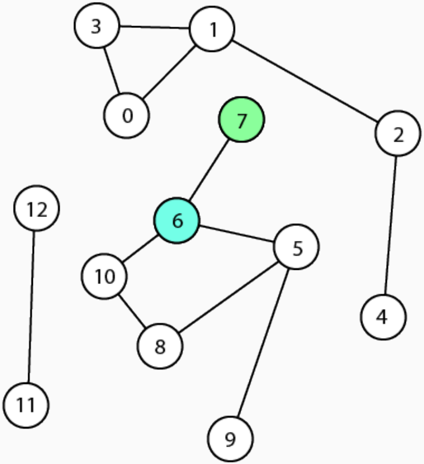
Generations / Queue

Generations:

[7]

[6]

[10, 5]



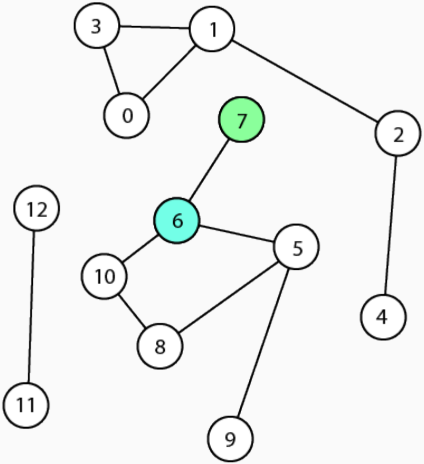
Generations / Queue

Generations:

[7]

[6]

[10, 5]



Generations / Queue

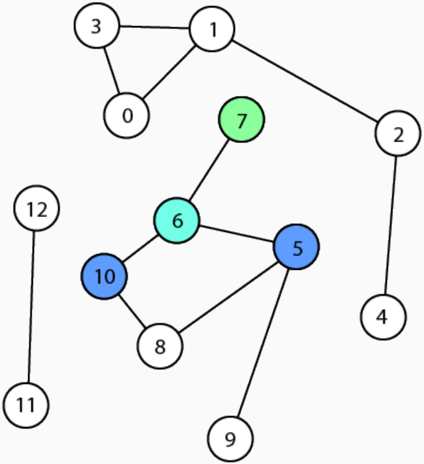
Generations:

[7]

[6]

[10, 5]

[ ]



Generations / Queue

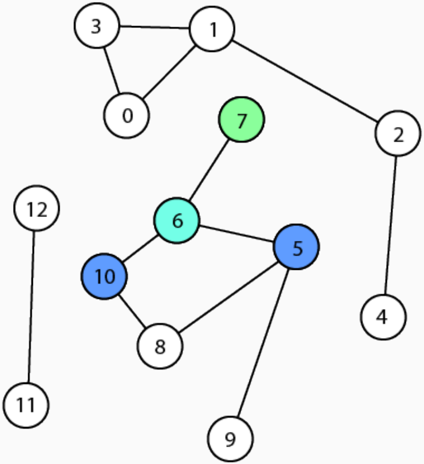
Generations:

[7]

[6]

[10, 5]

[8]



Generations / Queue

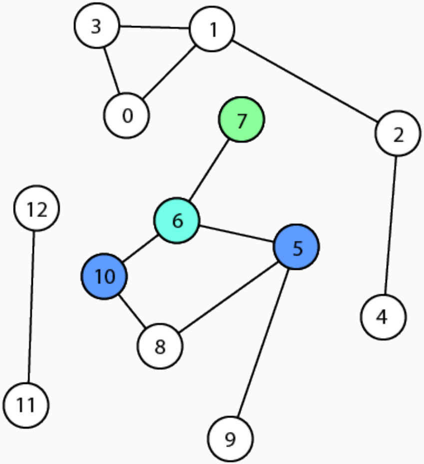
Generations:

[7]

[6]

[10, 5]

[8]



Generations / Queue

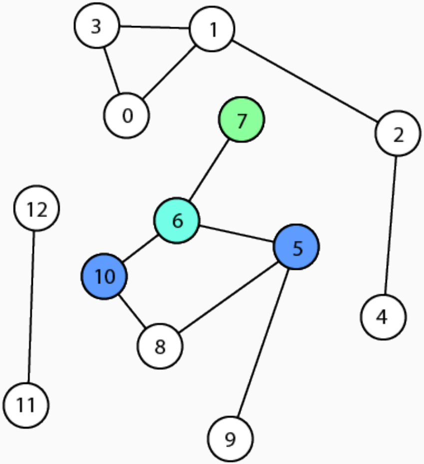
Generations:

[7]

[6]

[10, 5]

[8]



Generations / Queue

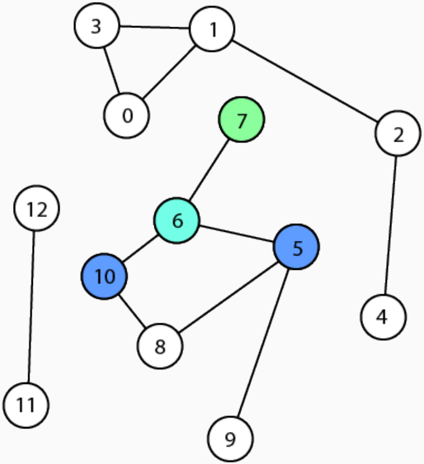
Generations:

[7]

[6]

[10, 5]

[8, 9]



Generations / Queue

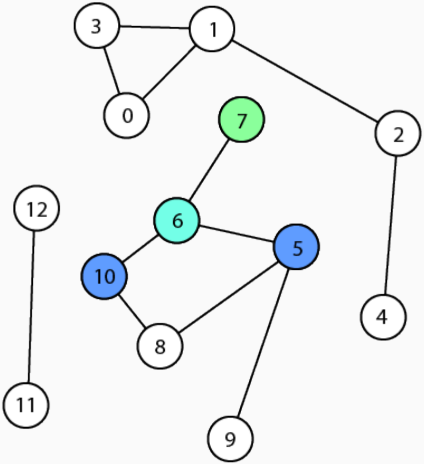
Generations:

[7]

[6]

[10, 5]

[8, 9]



Generations / Queue

Generations:

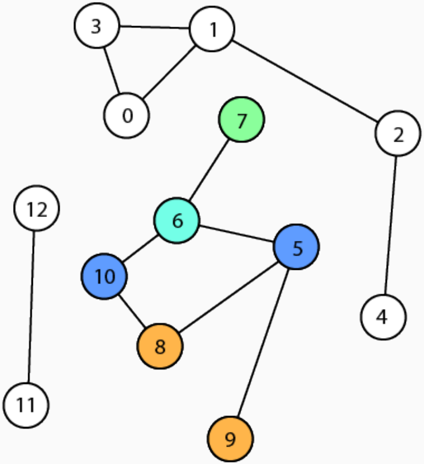
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations:

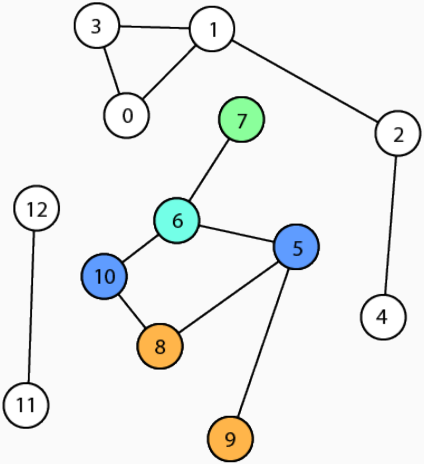
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations:

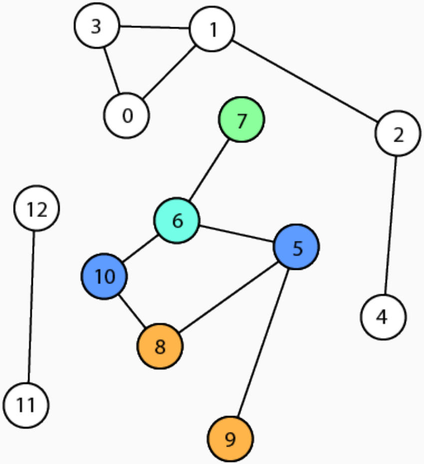
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations:

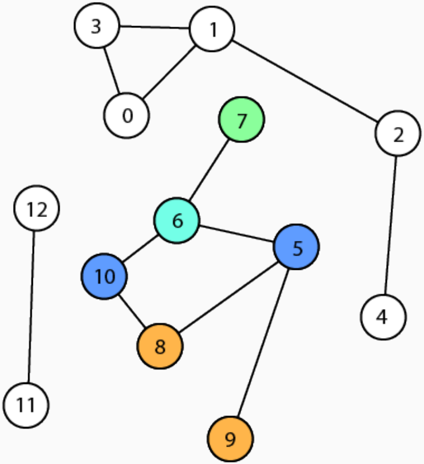
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations

[7]

[6]

[10, 5]

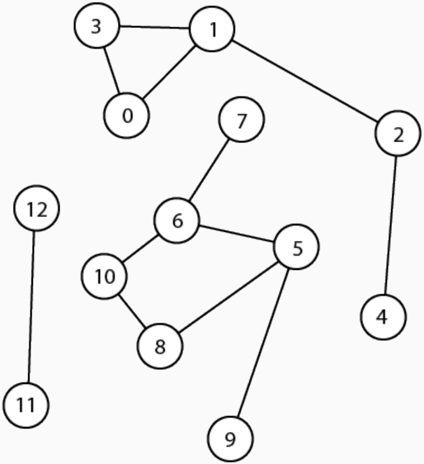
[8, 9]

Queue (first in, first out)

[7, 6, 10, 5, 8, 9]

Adjacency List

0: [1, 3]
1: [0, 2, 3]
2: [1, 4]
3: [0, 1]
4: [2]
5: [6, 8, 9]
6: [5, 7, 10]
7: [6]
8: [5, 10]
9: [5]
10: [6, 8]
11: [12]
12: [11]



BFS Implementation

- Visited flag for each node
- Queue to store neighbors
- `graph` is adjacency list

```
from collections import deque

def bfs(start):
  q = deque()
  visited = [False] * len(graph)
```

BFS Implementation

- Process start node

```
def bfs(start):
    ...
    visited[start] = True
    q.appendleft(start)
```

BFS Implementation

- Process neighbors
- Check whether not visited
- Add to queue and set visited

```
def bfs(start):
    ...
    while len(q) > 0:
        current = q.pop()
        for neighbor in graph[current]:
            if not visited[neighbor]:
                q.appendleft(neighbor)
                visited[neighbor] = True
```

BFS Implementation

- Return visited nodes

```python
def bfs(start, end):
    ...
    return visited
```

BFS Implementation

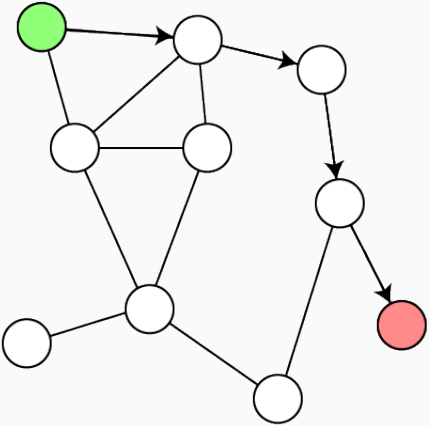
```python
def bfs(start):
    q = deque() # initialize
    visited = [False] * len(graph)

    visited[start] = True
    q.appendleft(start)

    while len(q) > 0: # traverse graph
        current = q.pop()
        for neighbor in graph[current]:
            if not visited[neighbor]:
                q.appendleft(neighbor)
                visited[neighbor] = True
    return visited
```

Shortest Distance Implementation

Shortest Distance



Shortest Distance Implementation

1. List of distance to start
2. Upon adding to queue, store distance

Shortest Distance Implementation

- List of distance to start

```
def bfs(start, target):
    ...
    visited = [False] * len(graph)
    distance = [0] * len(graph)
    ...
```

Shortest Distance Implementation

- Store distance

```
def bfs(start, target):
    ...
    visited[neighbor] = True
    distance[neighbor] = distance[current] + 1
    ...
```

Shortest Distance Implementation

- Return shortest distance
- Note: we know shortest distance to every node

```python
def bfs(start, target):
    ...
    return distance[target]
```

Shortest Distance Implementation

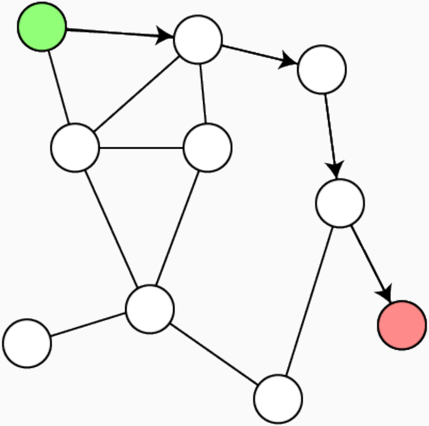
```python
def bfs(start, target):
  q = deque() # initialize
  visited = [False] * len(graph)
  distance = [0] * len(graph)

  visited[start] = True
  q.appendleft(start)

  while len(q) > 0: # traverse graph
    current = q.pop()
    for neighbor in graph[current]:
      if not visited[neighbor]:
        q.appendleft(neighbor)
        visited[neighbor] = True
        distance[neighbor] = distance[current] + 1
  return distance[target]
```

Shortest Path Implementation

Shortest Path



Shortest Path Implementation

1. Store ”parent” of node
2. Upon adding to queue, store parent

Shortest Path Implementation

- List of parents

```
def bfs(start, target):
    ...
    visited = [False] * len(graph)
    parent = [-1] * len(graph)
    ...
```

Shortest Path Implementation

- Store parent

```
def bfs(start, target):
    ...
    visited[neighbor] = True
    parent[neighbor] = current
    ...
```

Shortest Path Implementation

- Return shortest path
- Note: we know shortest path to every node

```python
def bfs(start, target):
    ...
    if parent[target] == -1:
        return []
    path = []
    current = target
    while current != -1:
        path.append(current)
        current = parent[current]
    return reversed(path)
```

Shortest Path Implementation

```python
def bfs(start, target):
  q = deque() # initialize
  visited = [False] * len(graph)
  parent = [-1] * len(graph)

  visited[start] = True
  q.appendleft(start)

  while len(q) > 0: # traverse graph
    current = q.pop()
    for neighbor in graph[current]:
      if not visited[neighbor]:
        q.appendleft(neighbor)
        visited[neighbor] = True
        parent[neighbor] = current

  if parent[target] == -1: # reconstruct path
    return []
  path = []
  current = target
  while current != -1:
    path.append(current)
    current = parent[current]
  return list(reversed(path))
```

Runtime

Runtime

- Each node is visited exactly once
- Each edge is visited exactly twice

Runtime

$$\mathcal{O}(n + m)$$

Summary

Advanced

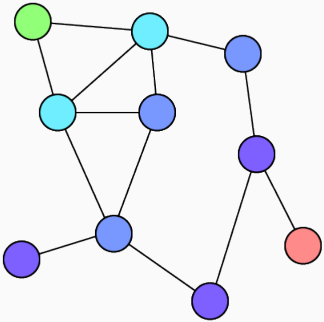
- Works with directed graphs
- Can find shortest path to all nodes
- Sometimes only implicit state

Summary

- Similar to DFS
- Progress "by generation"
- Useful for many different problems
- E.g. components, shortest path, bipartite

Conclusion

Bug Bounty: 1 Prügeli
(expires Sunday 10pm)



And now is your last chance for Pizza!