BFS

Breadth First Search

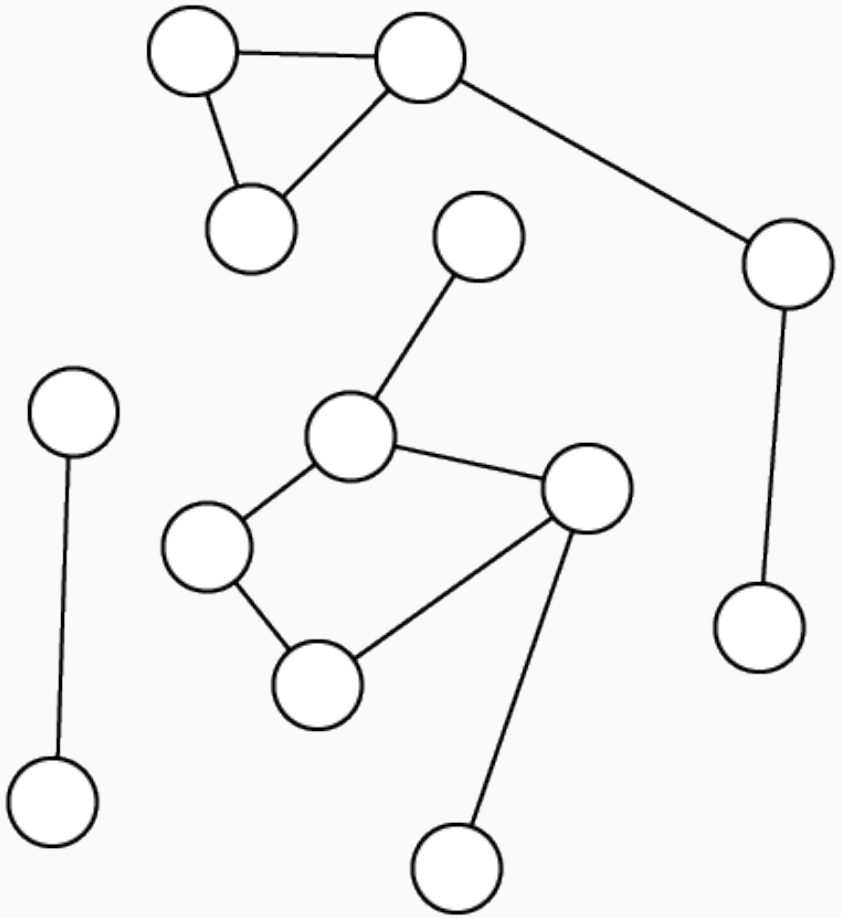
Benjamin Schmid

2019-11-10

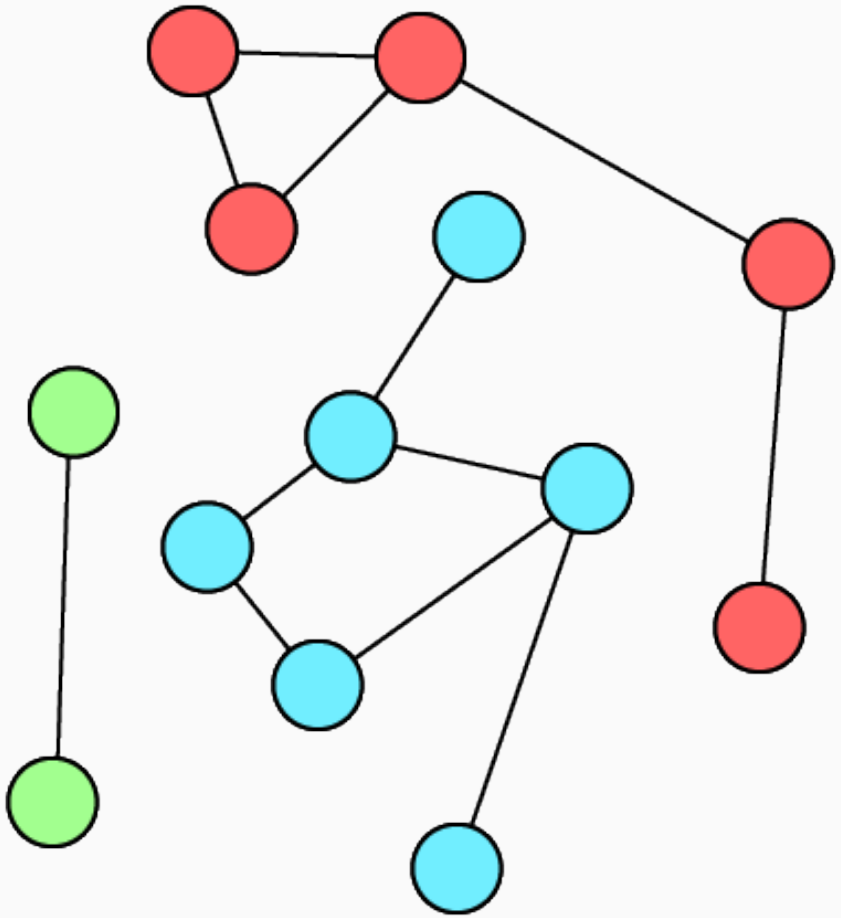
Swiss Olympiad in Informatics

Introduction

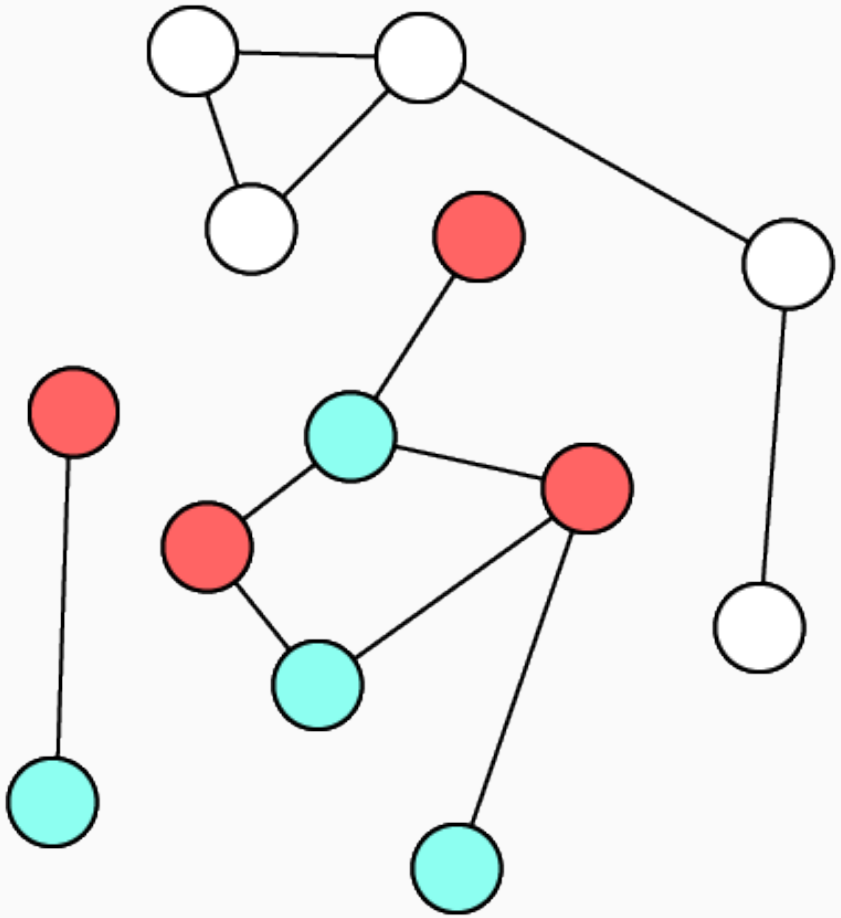
Graph traversal



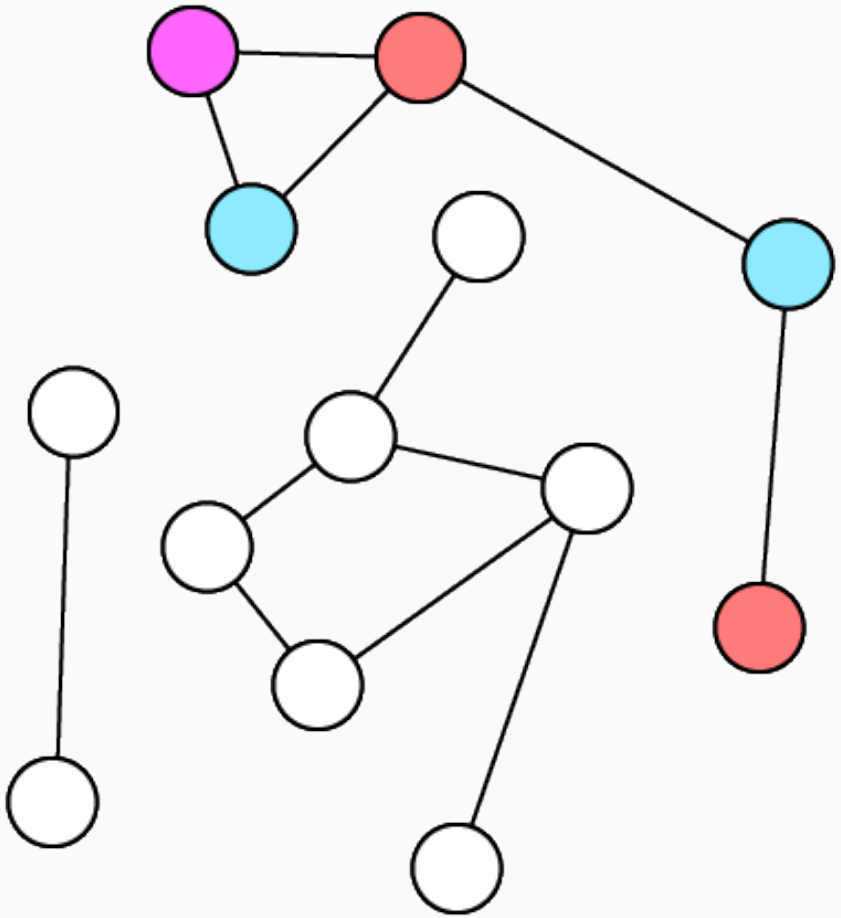
Components



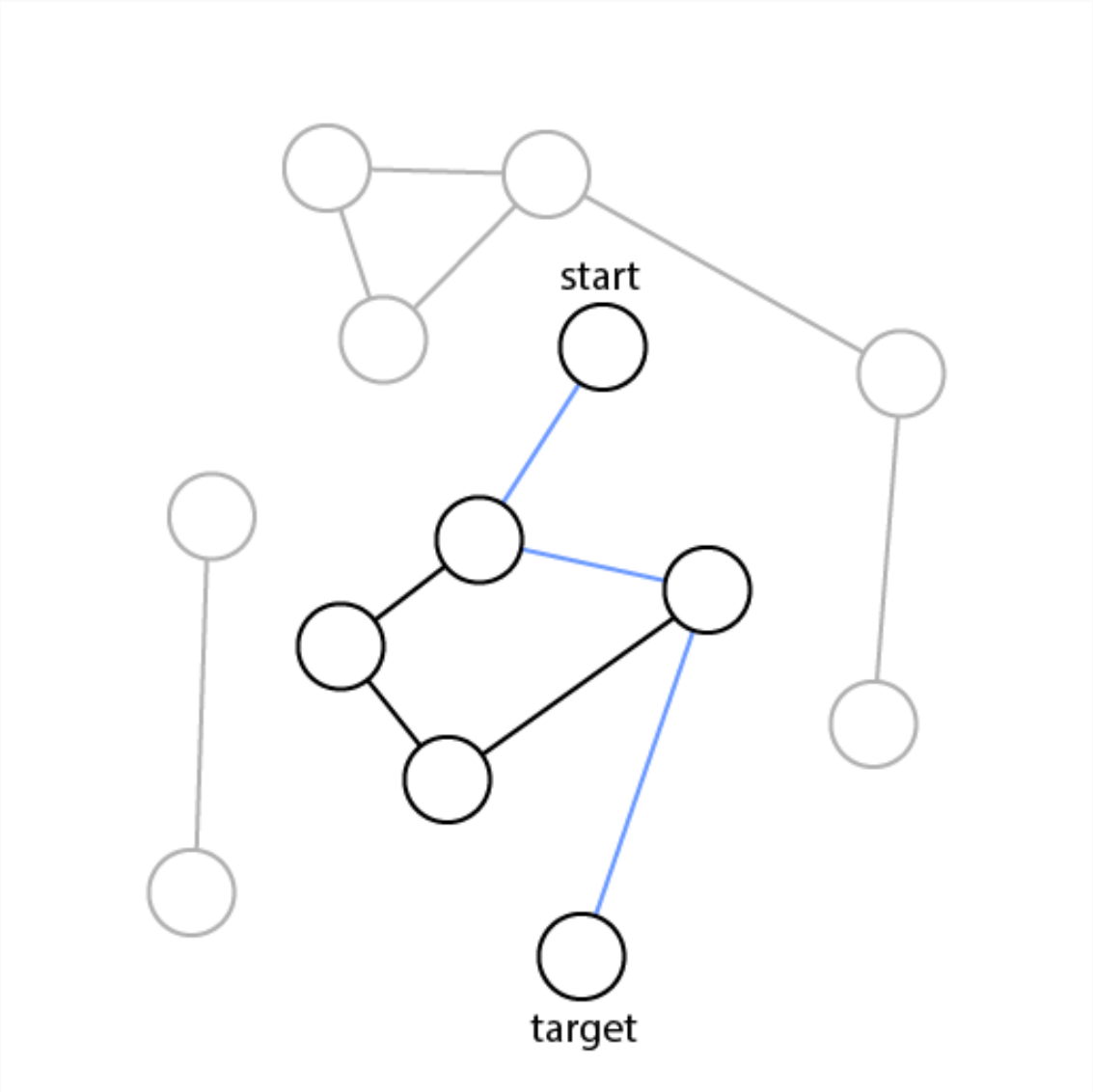
Bipartite



Bipartite

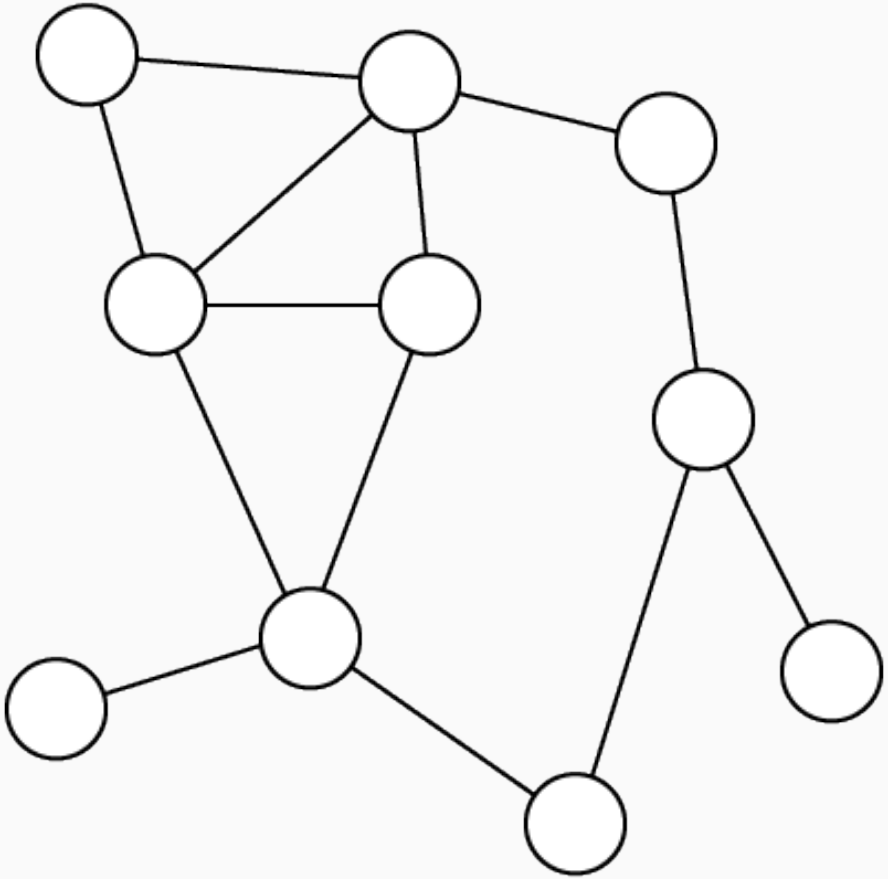


Shortest path

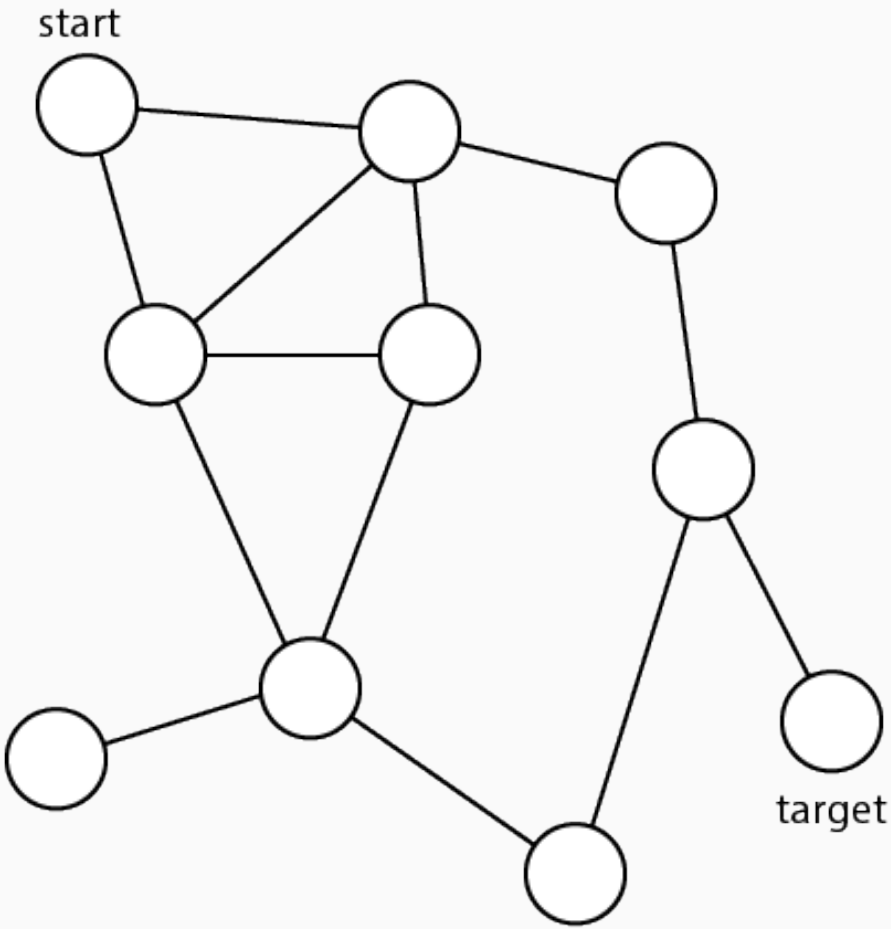


Shortest path

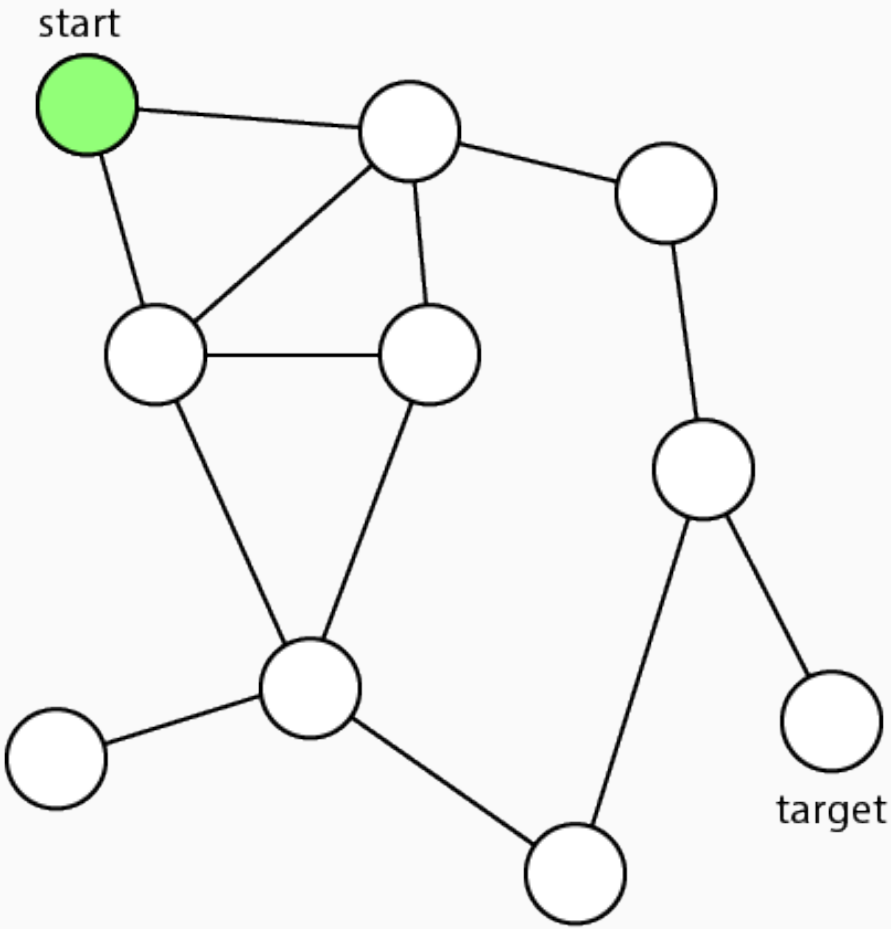
Shortest path



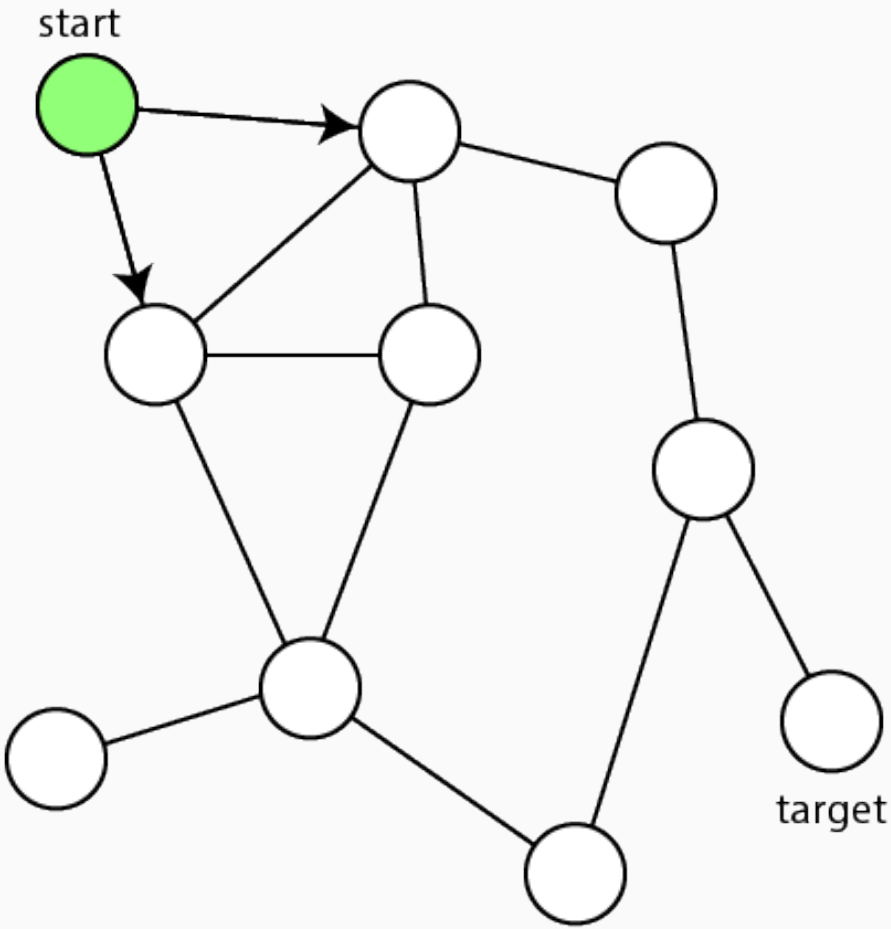
Shortest path



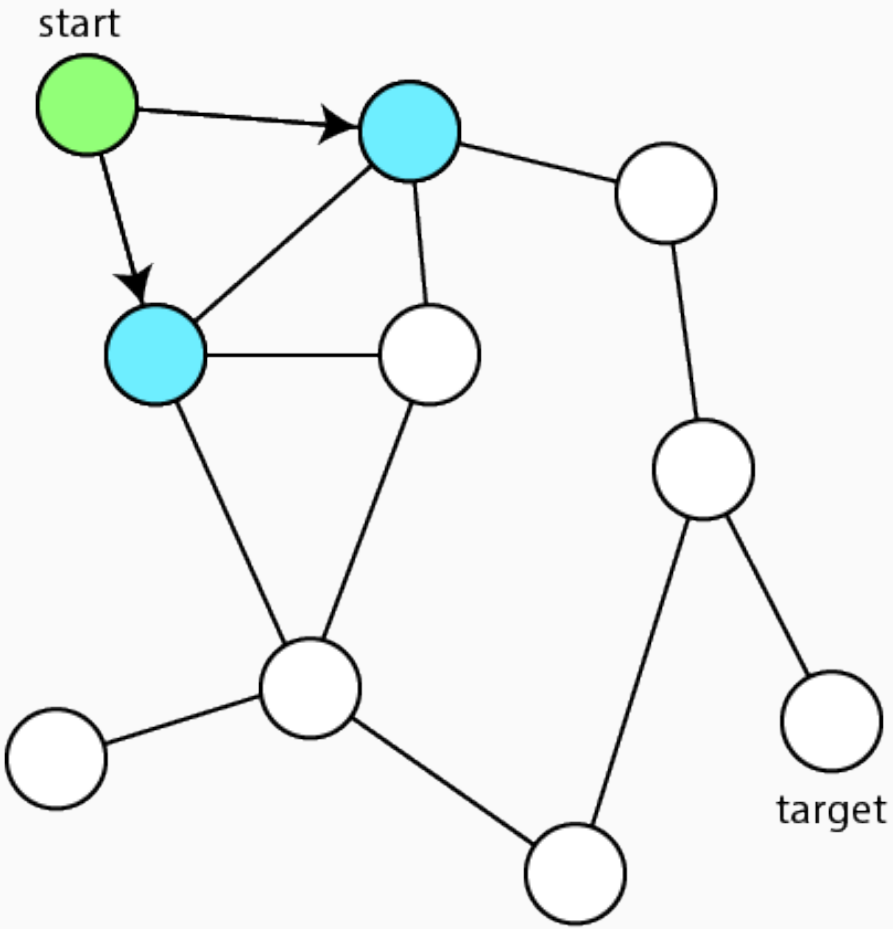
Shortest path



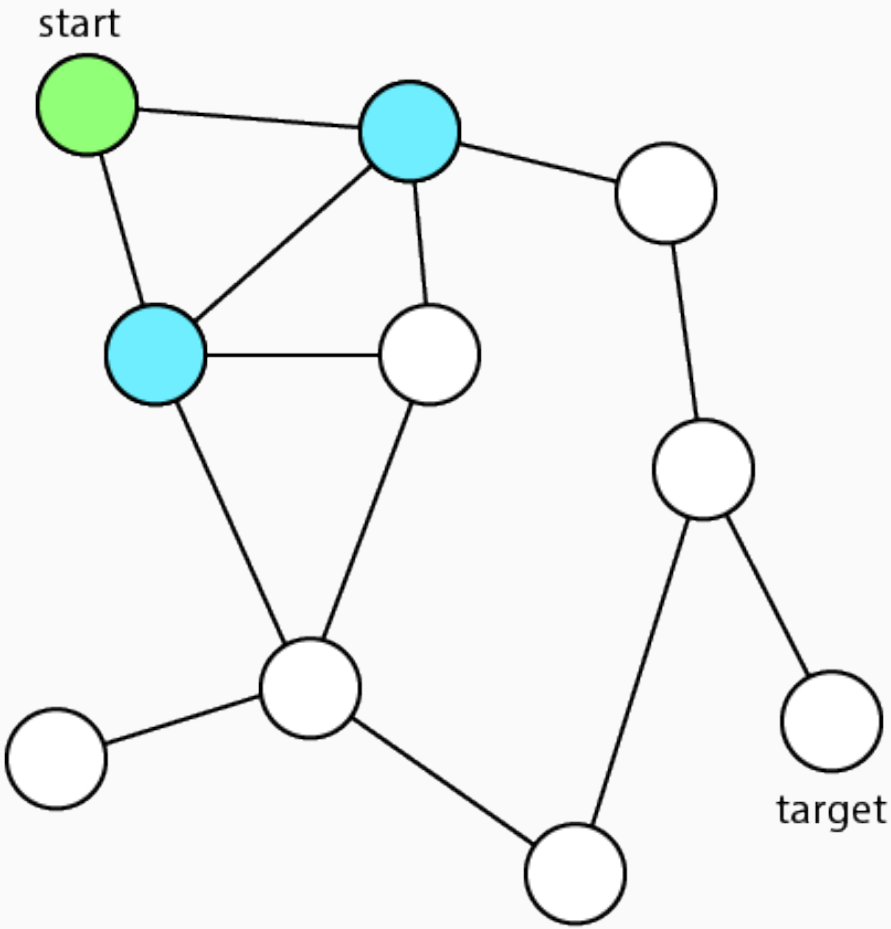
Shortest path



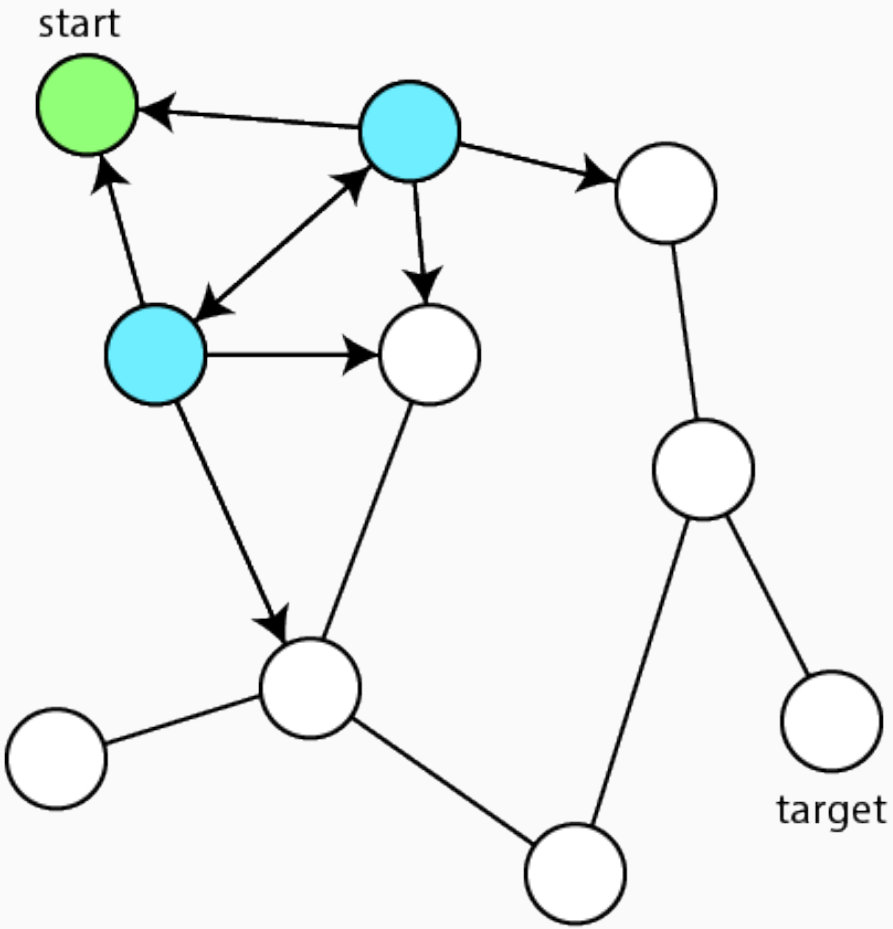
Shortest path



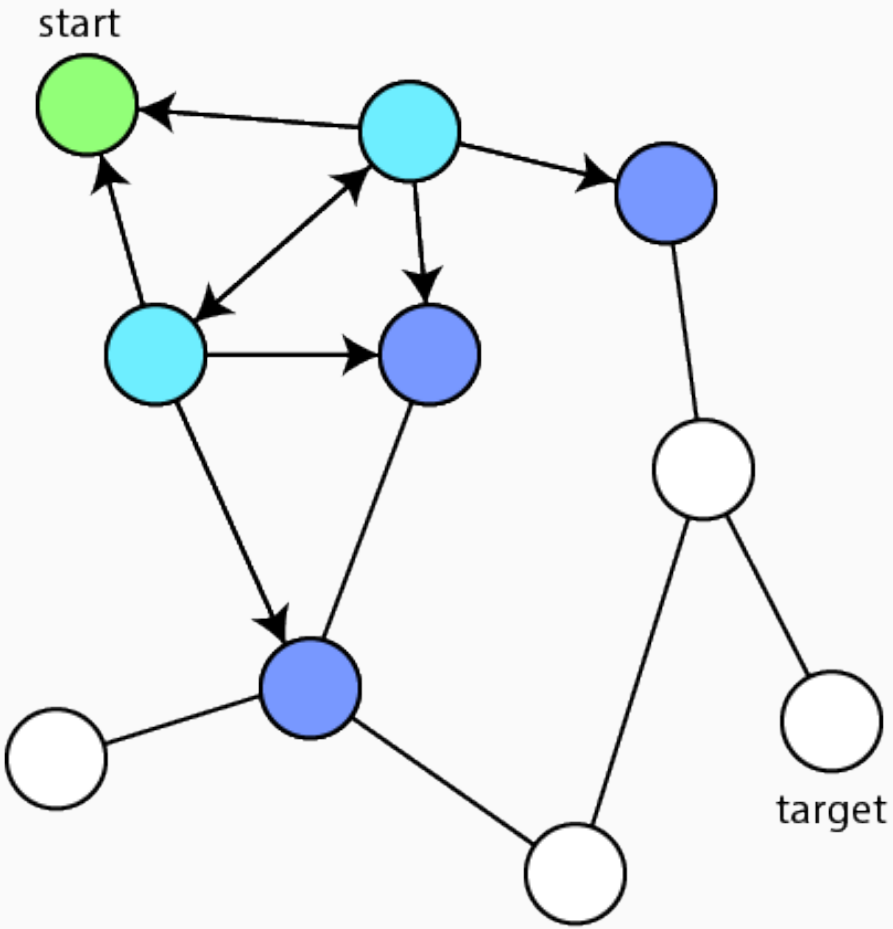
Shortest path



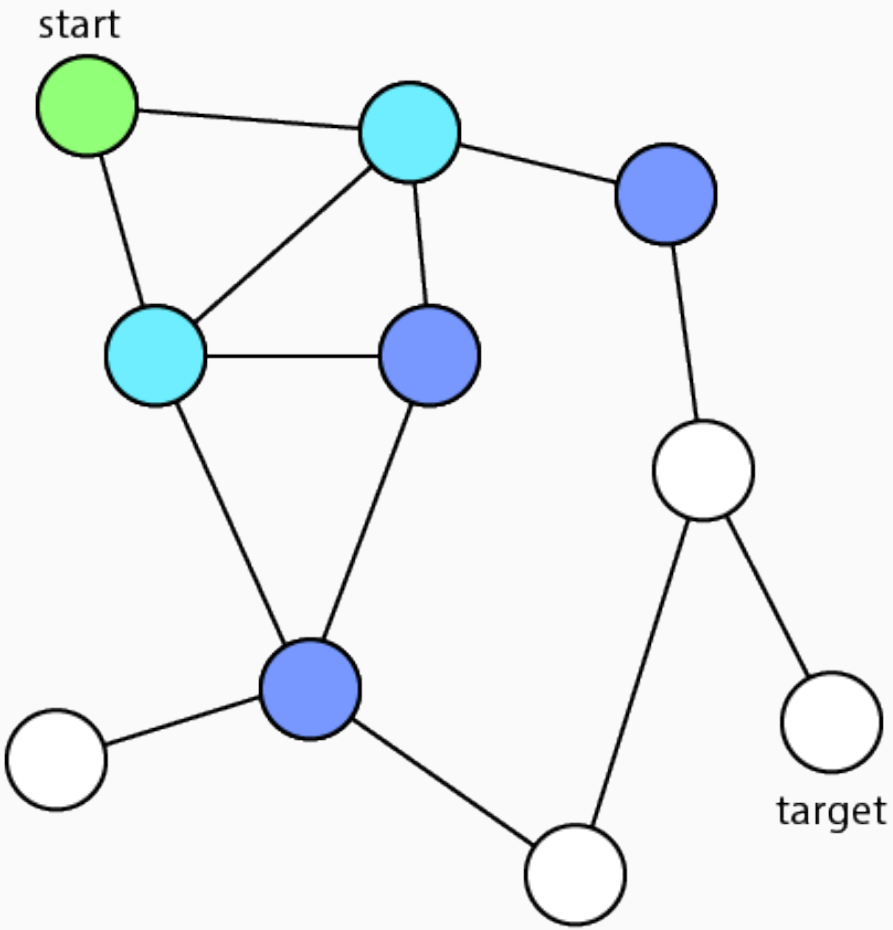
Shortest path



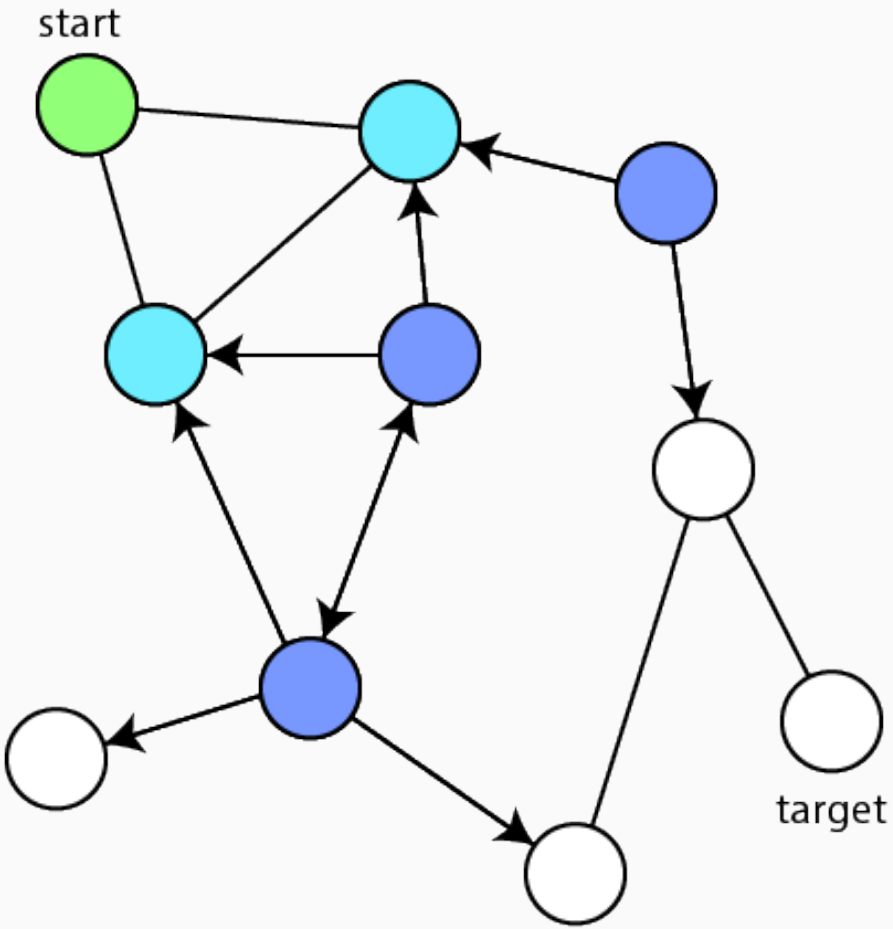
Shortest path



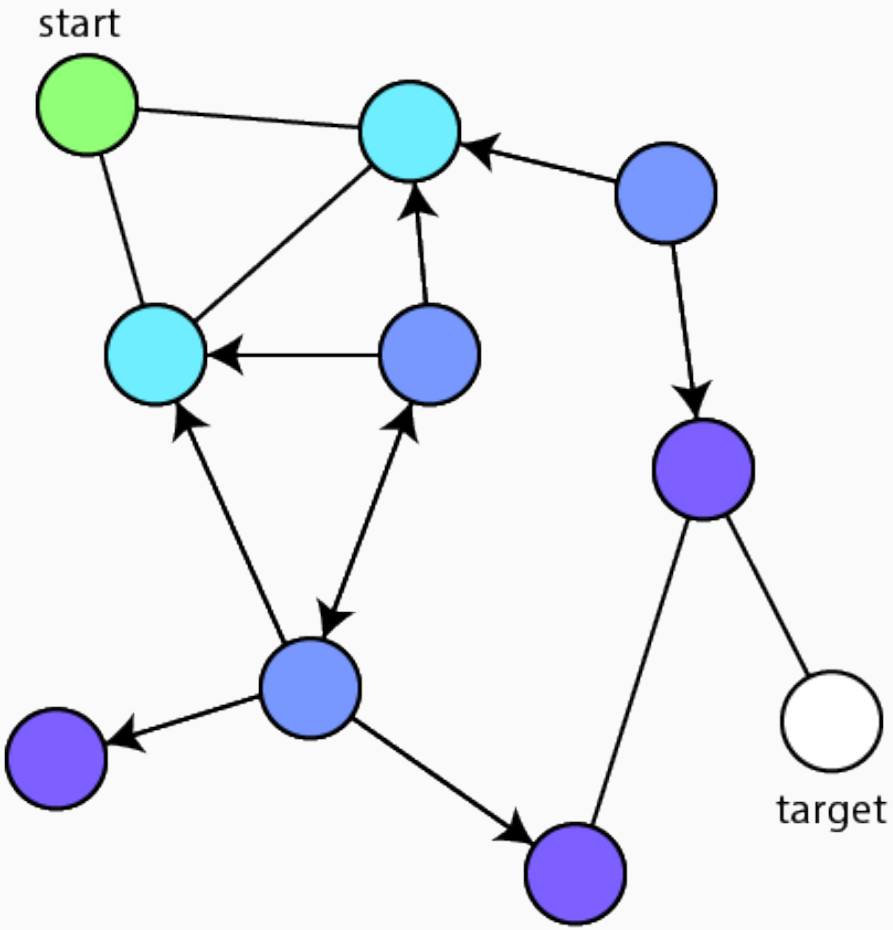
Shortest path



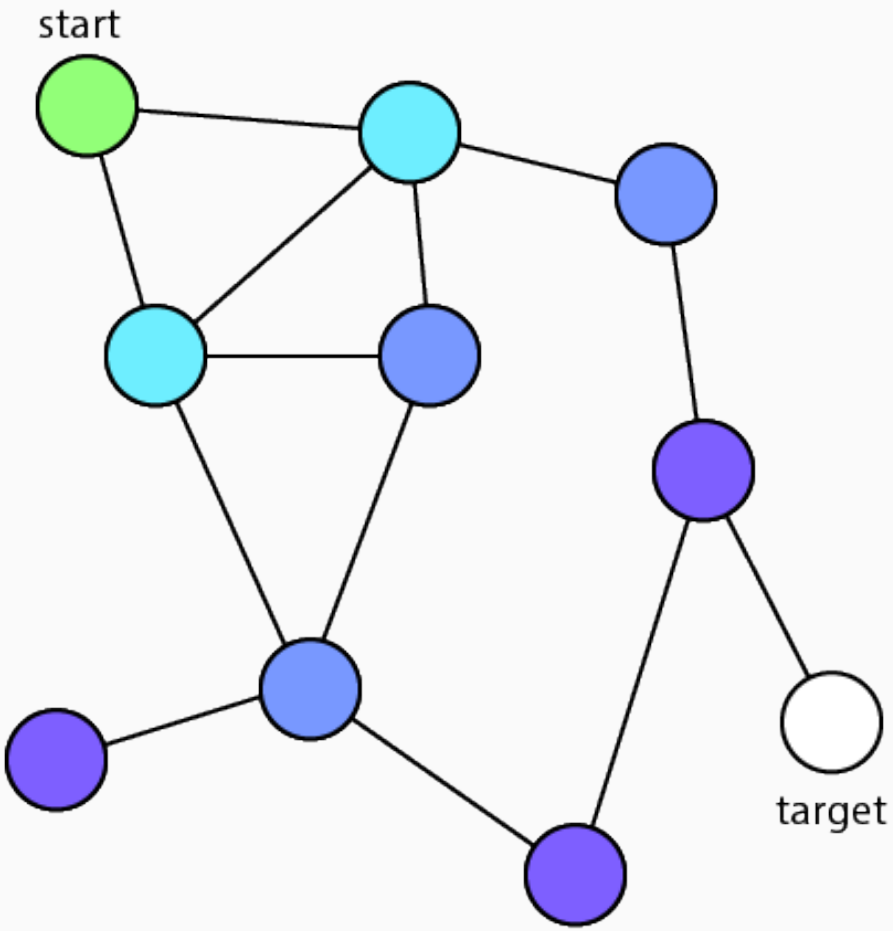
Shortest path



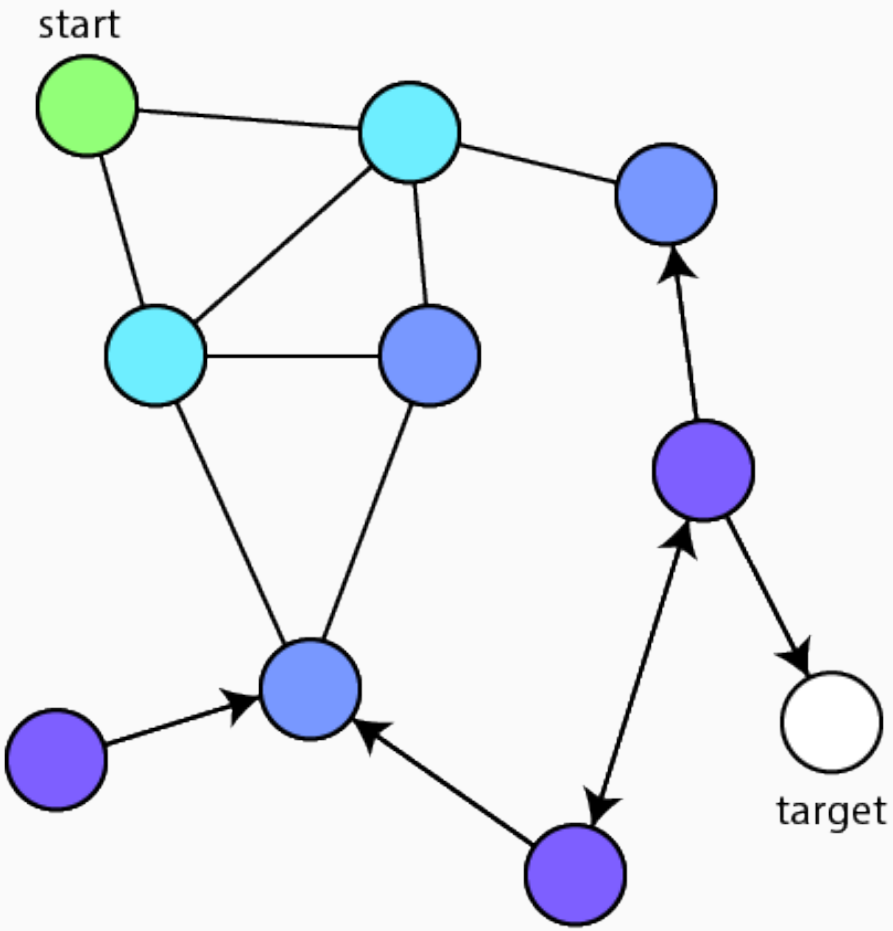
Shortest path



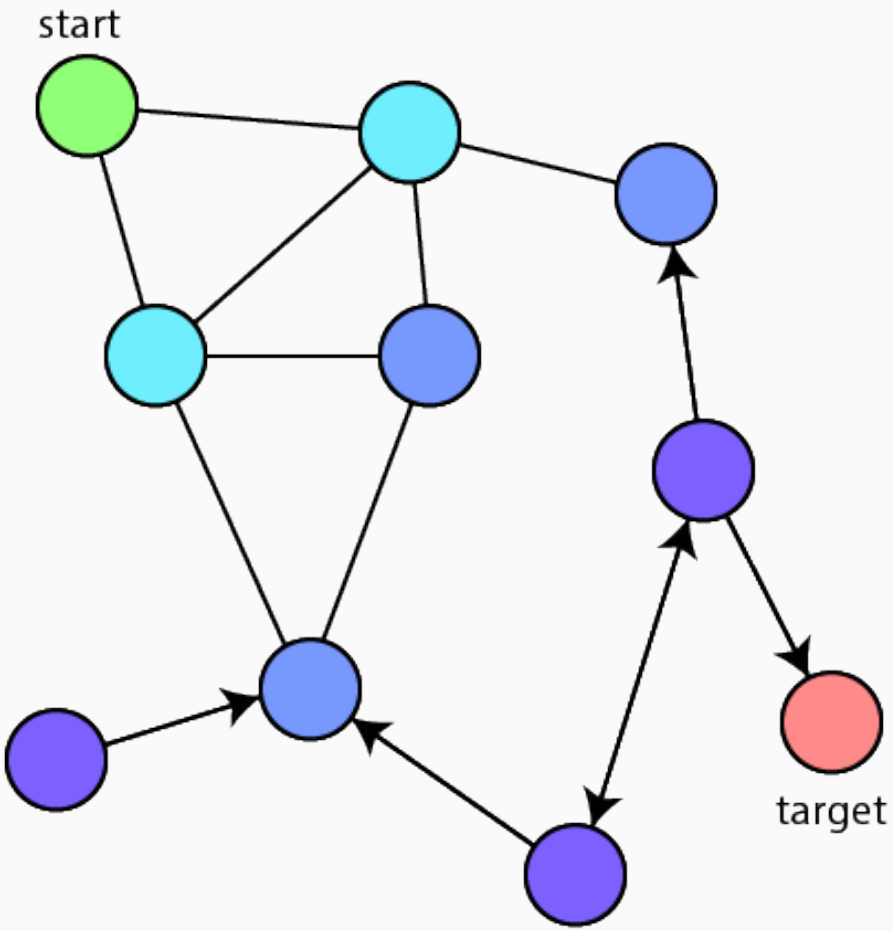
Shortest path



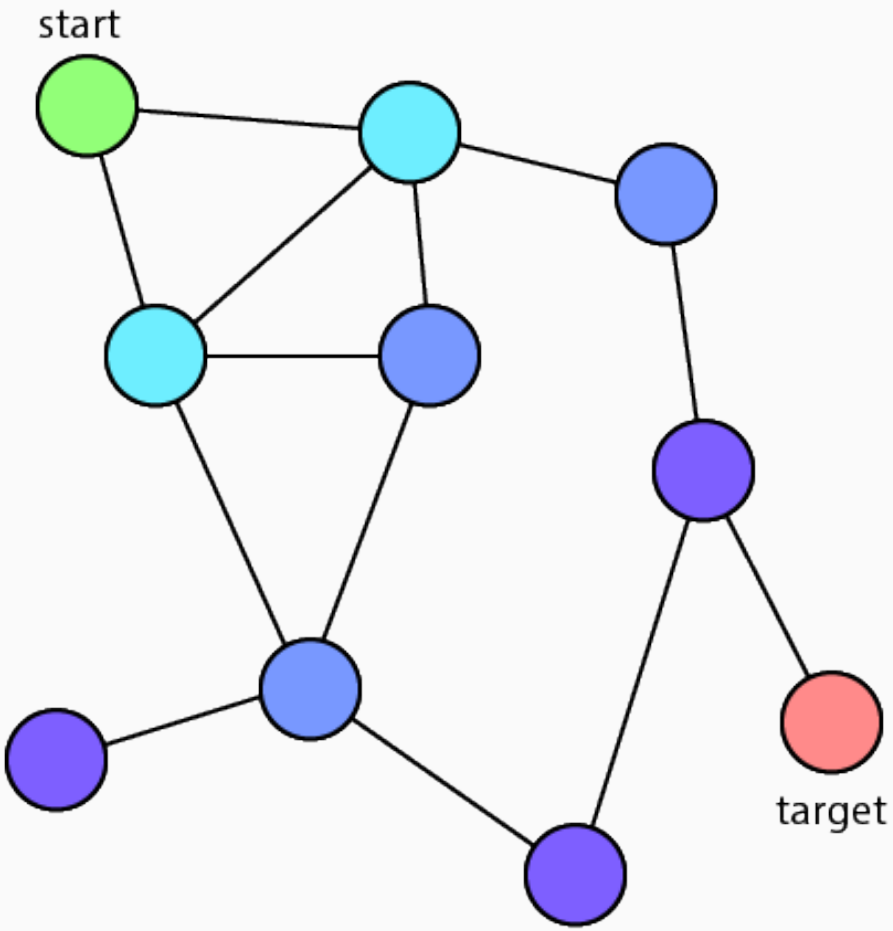
Shortest path



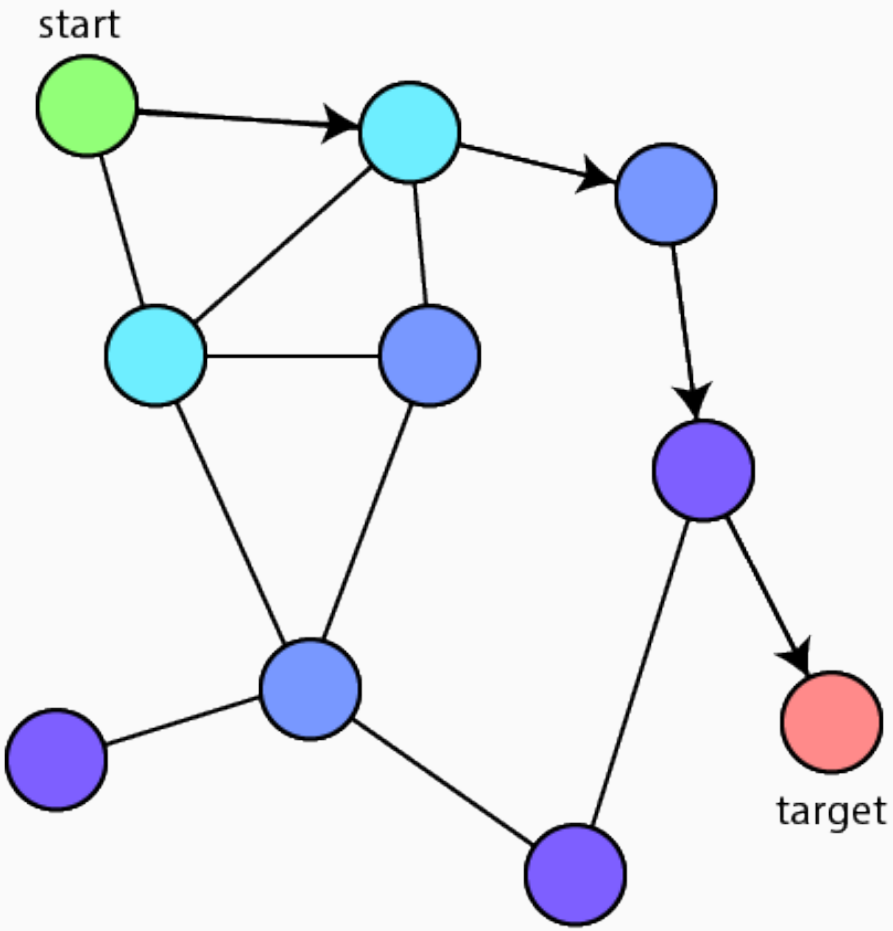
Shortest path



Shortest path



Shortest path

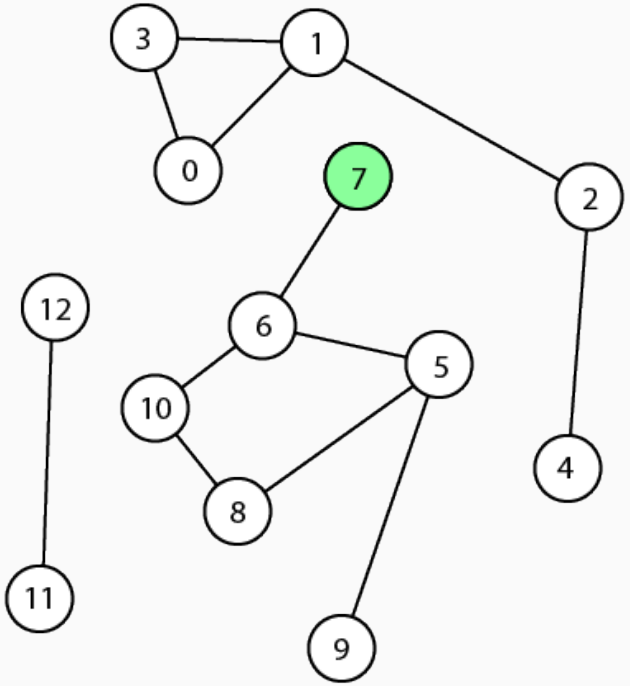


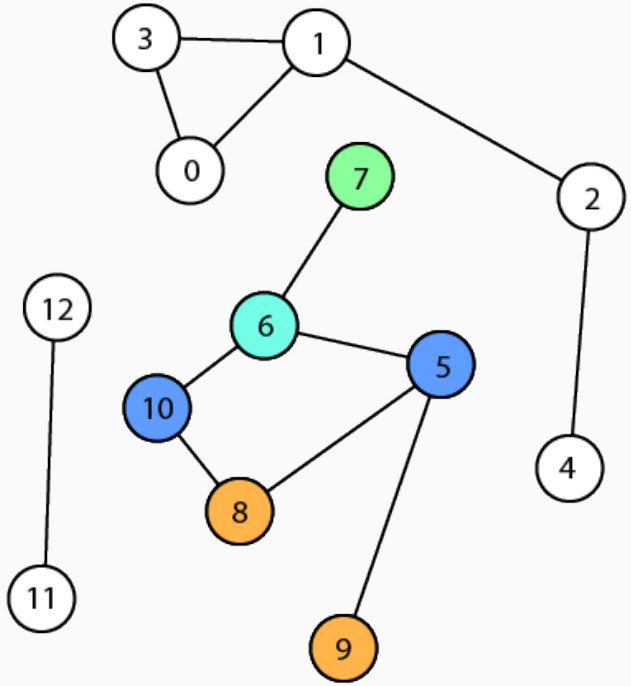
BFS

- 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

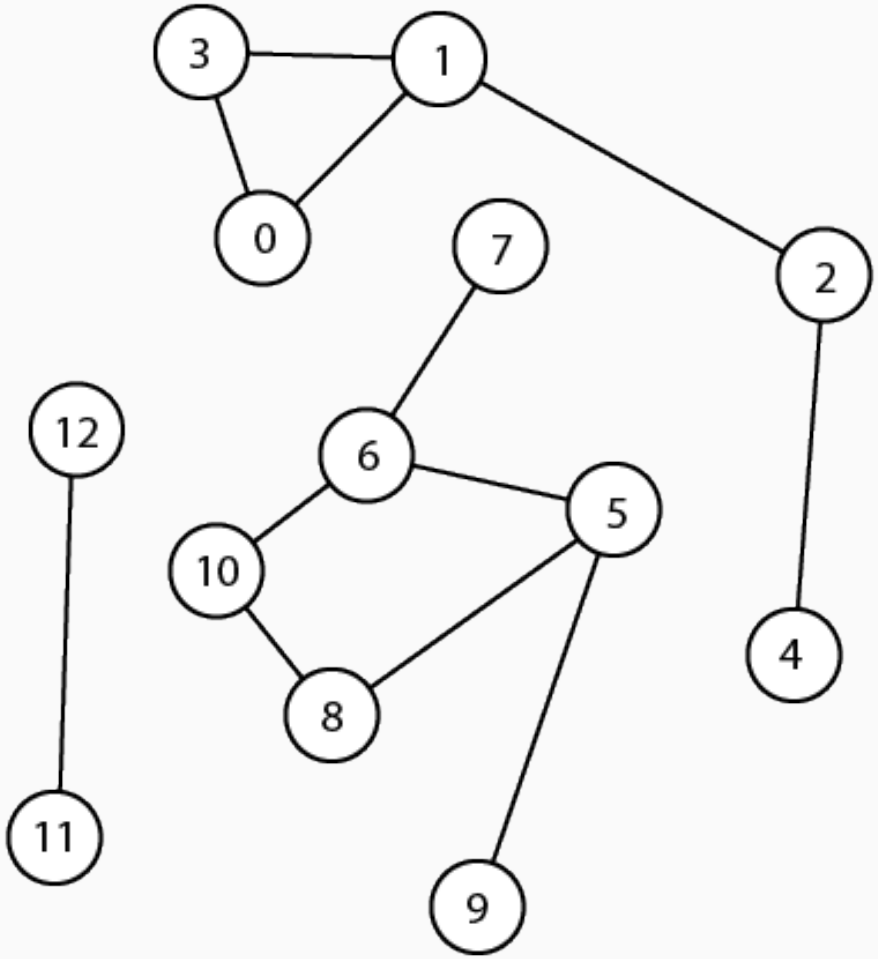




Generations / Queue

Generations:

[ ]

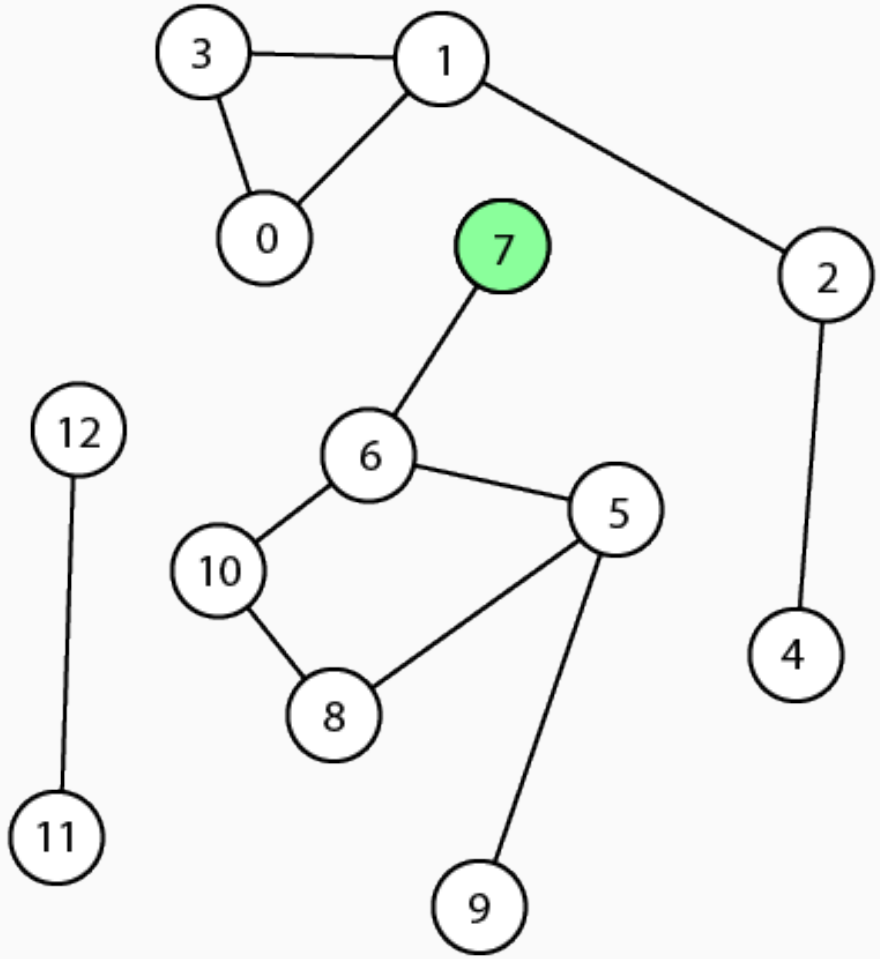


Generations / Queue

Generations:

[7]

[ ]

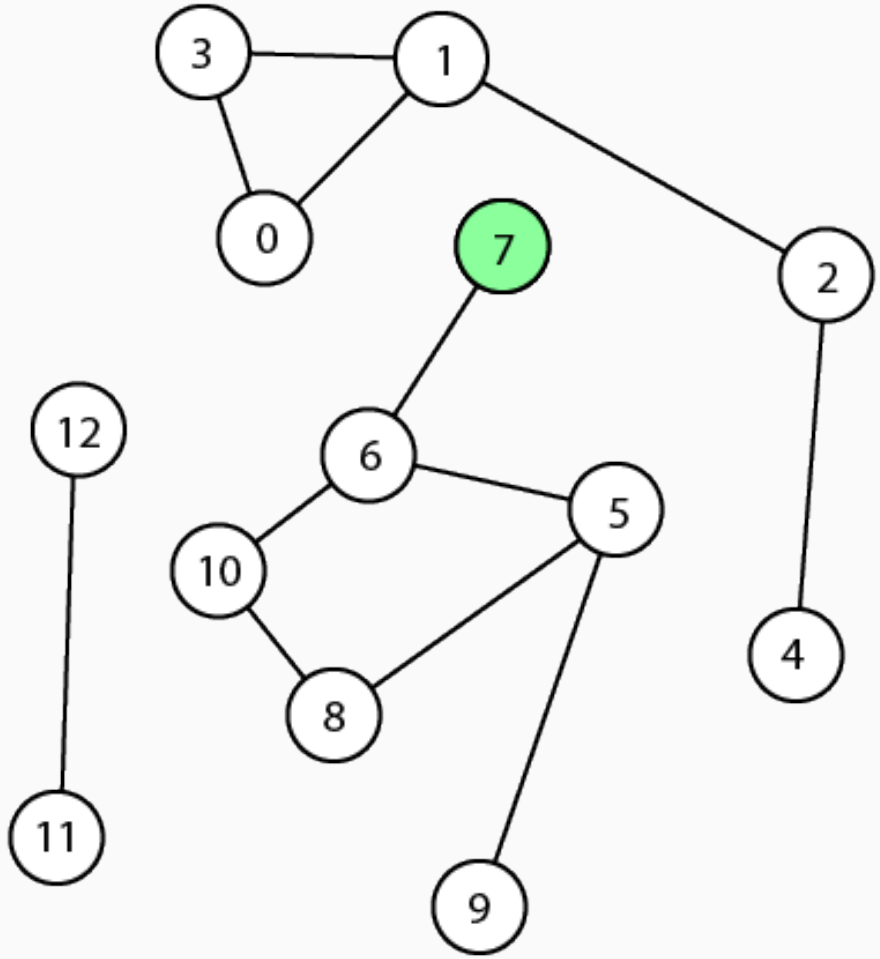


Generations / Queue

Generations:

[7]

[6]

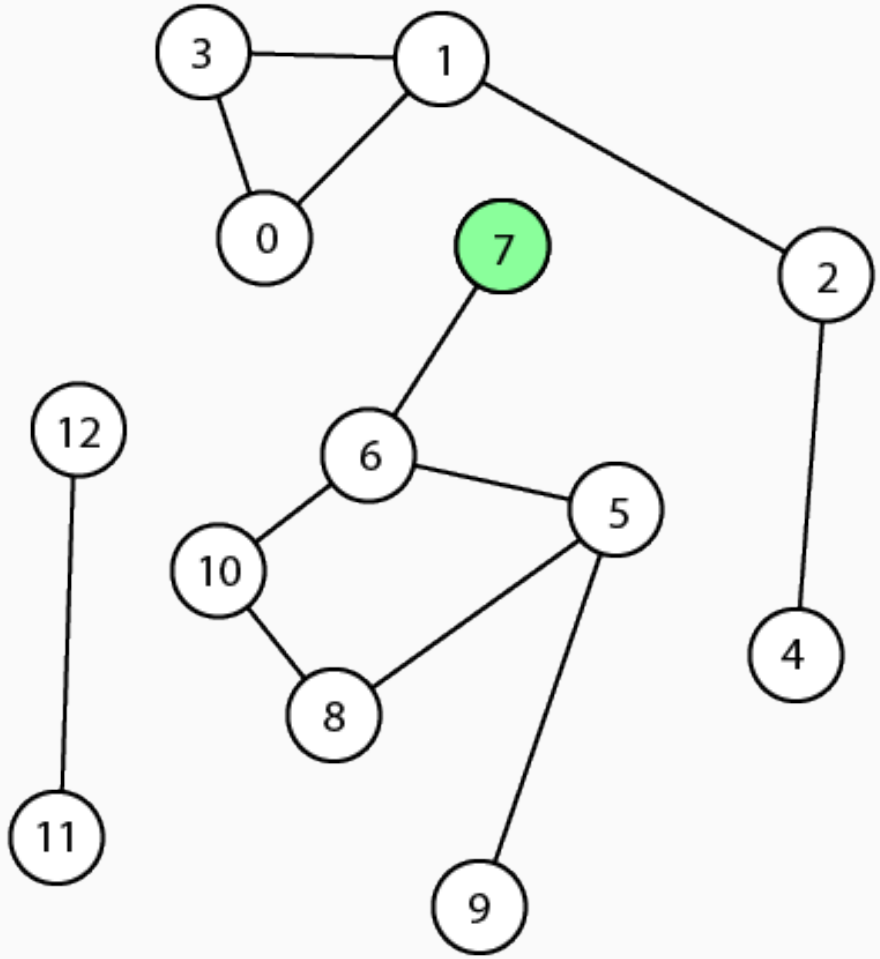


Generations / Queue

Generations:

[7]

[6]



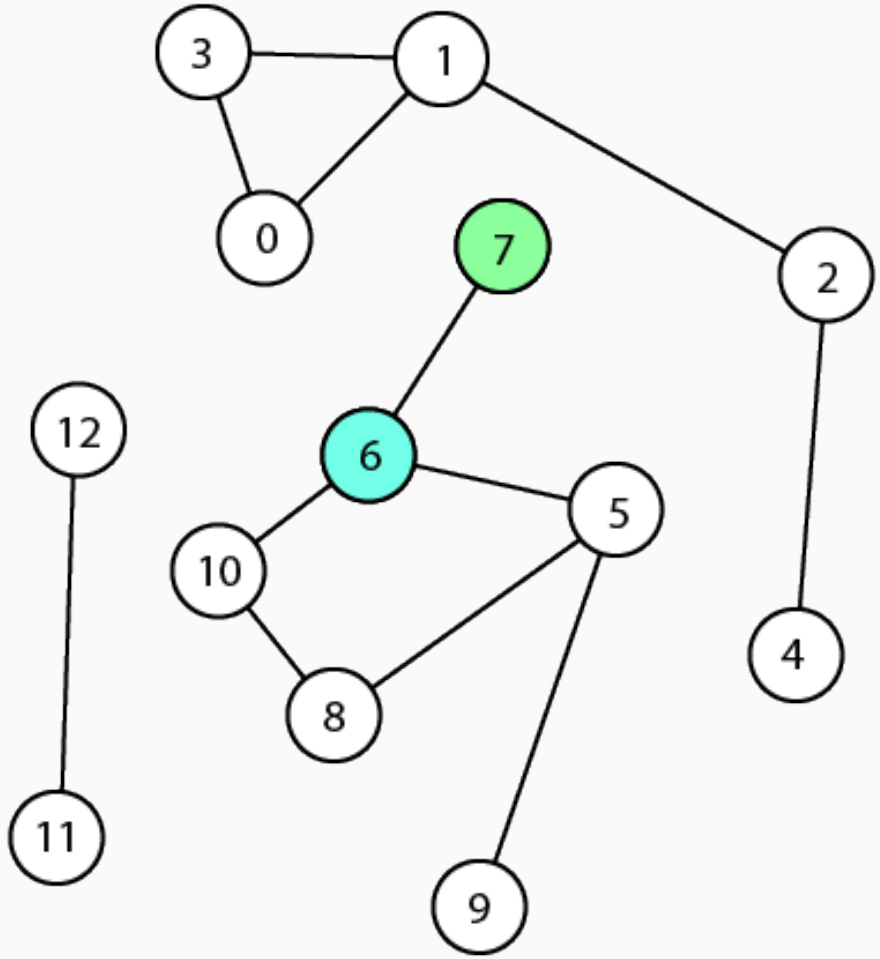
Generations / Queue

Generations:

[7]

[6]

[ ]



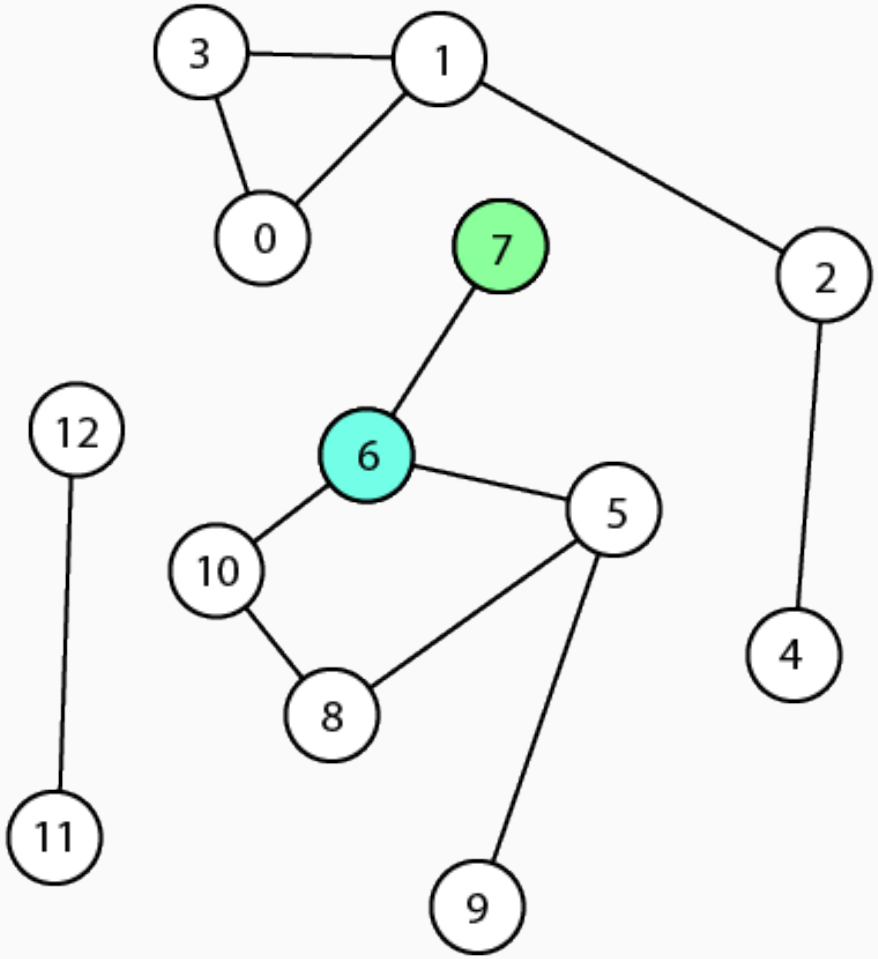
Generations / Queue

Generations:

[7]

[6]

[10, 5]



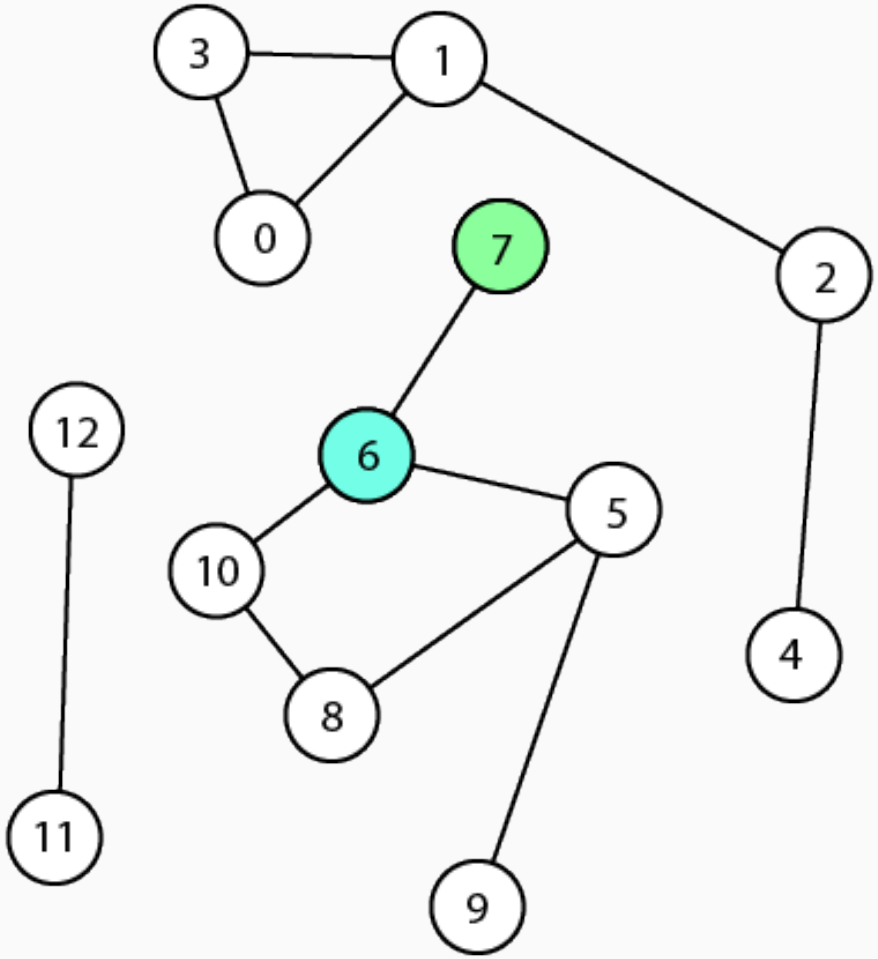
Generations / Queue

Generations:

[7]

[6]

[10, 5]



Generations / Queue

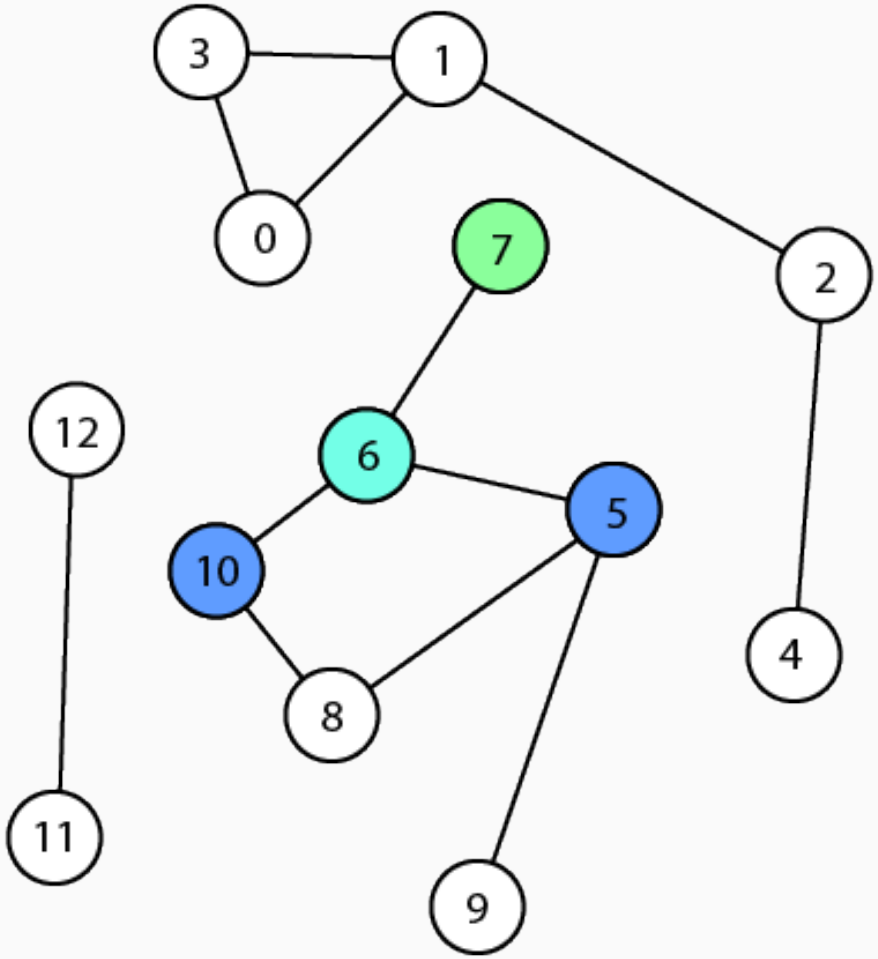
Generations:

[7]

[6]

[10, 5]

[ ]



Generations / Queue

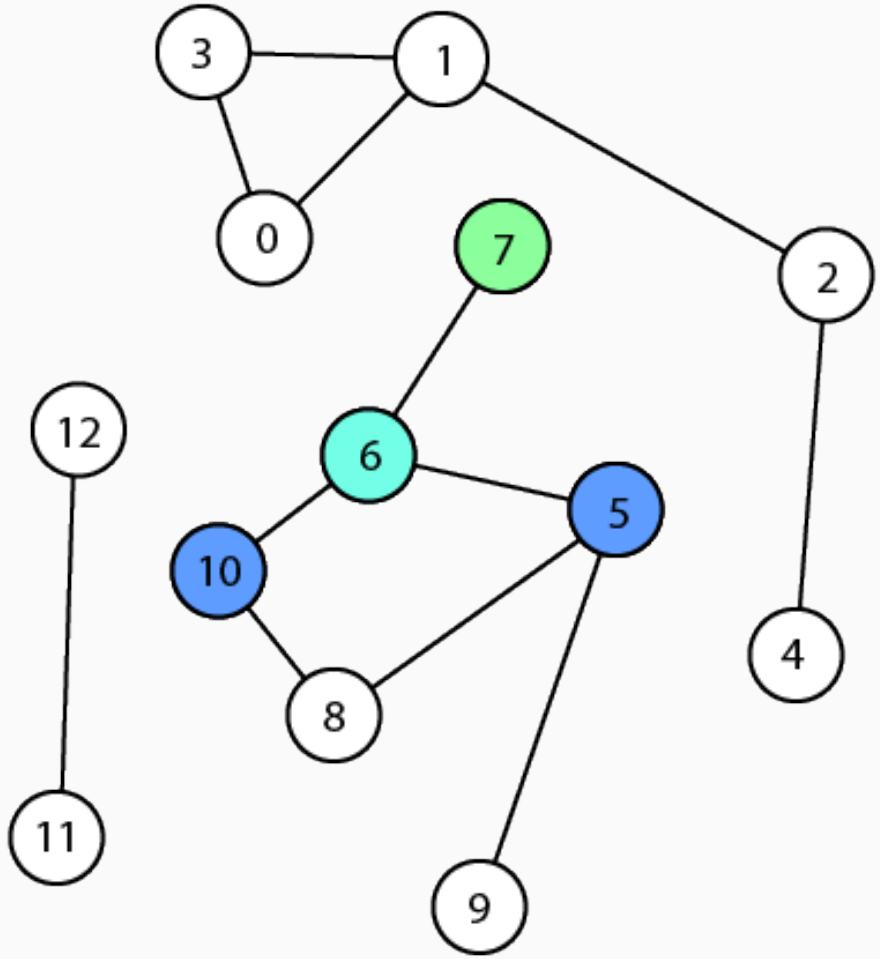
Generations:

[7]

[6]

[10, 5]

[8]



Generations / Queue

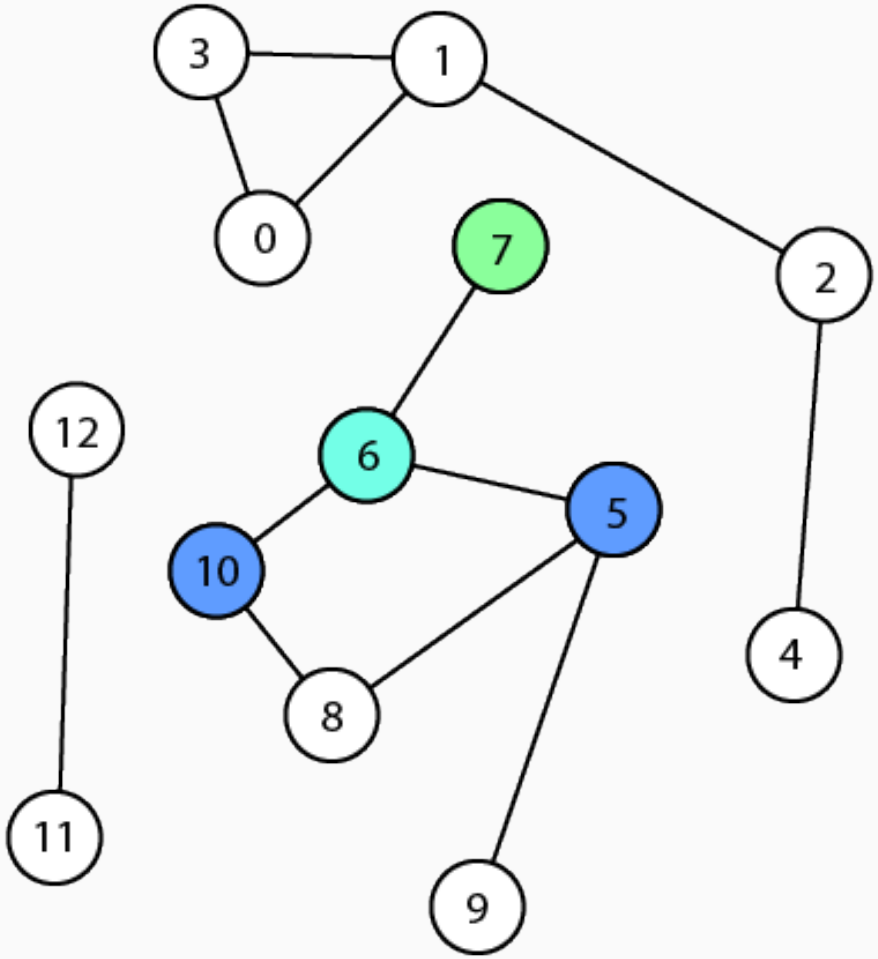
Generations:

[7]

[6]

[10, 5]

[8]



Generations / Queue

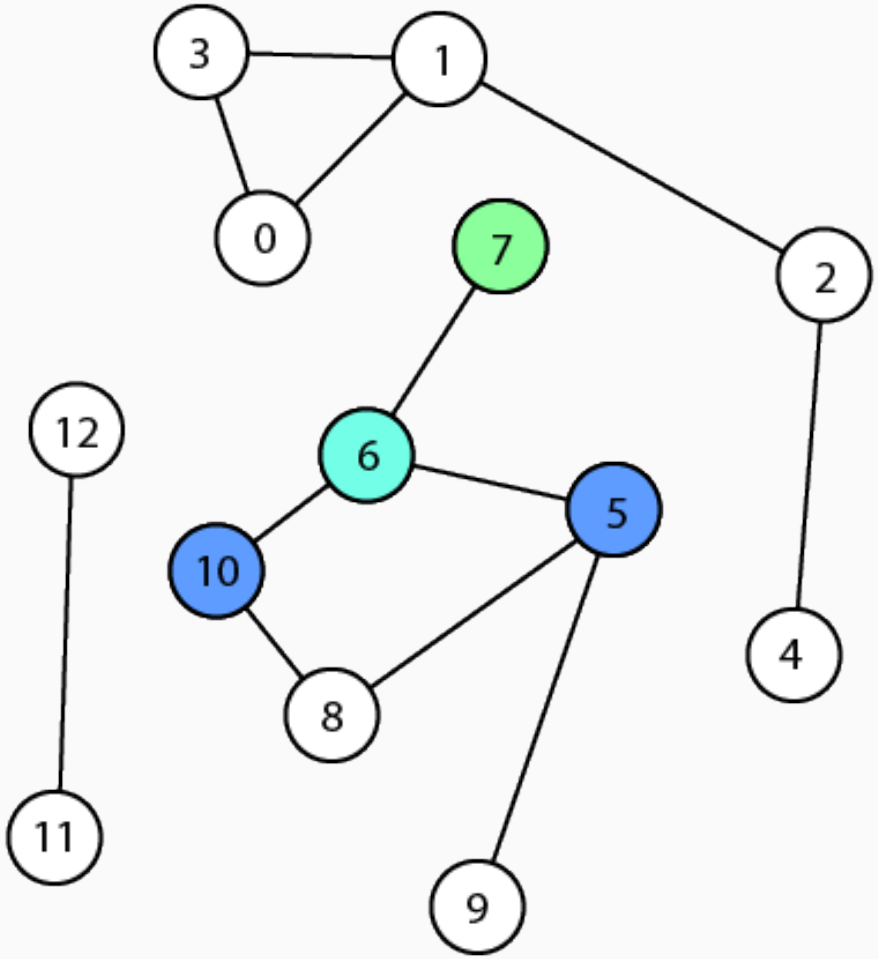
Generations:

[7]

[6]

[10, 5]

[8]



Generations / Queue

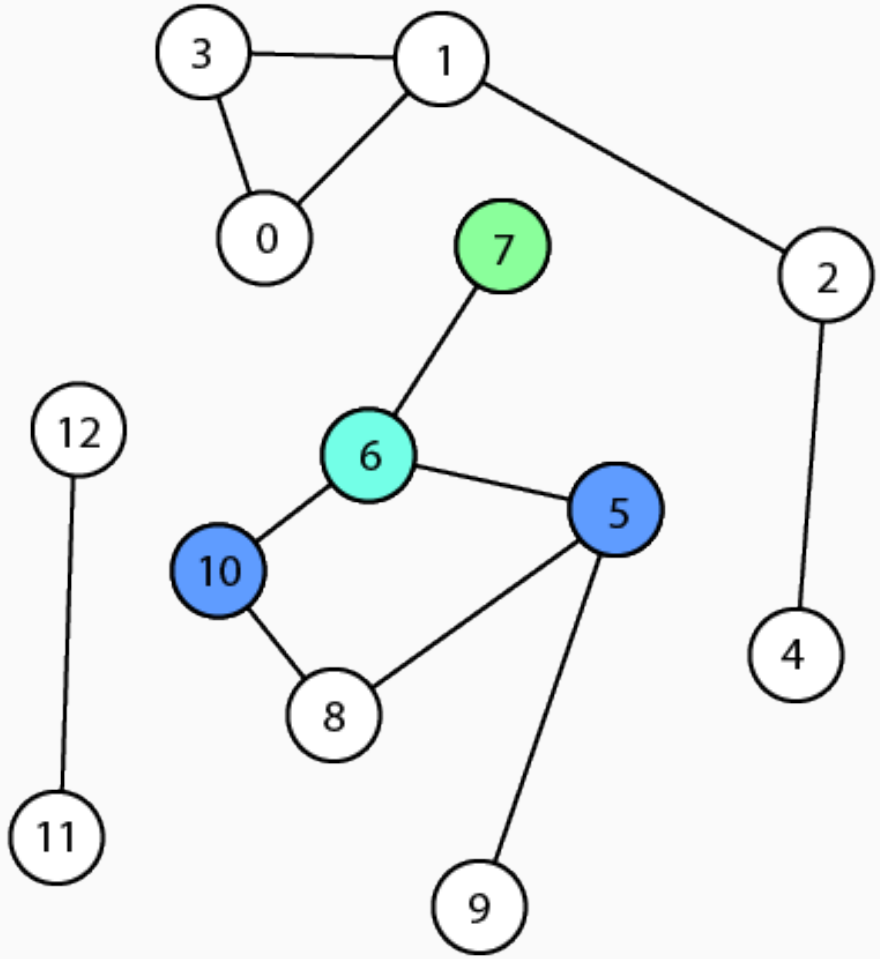
Generations:

[7]

[6]

[10, 5]

[8, 9]



Generations / Queue

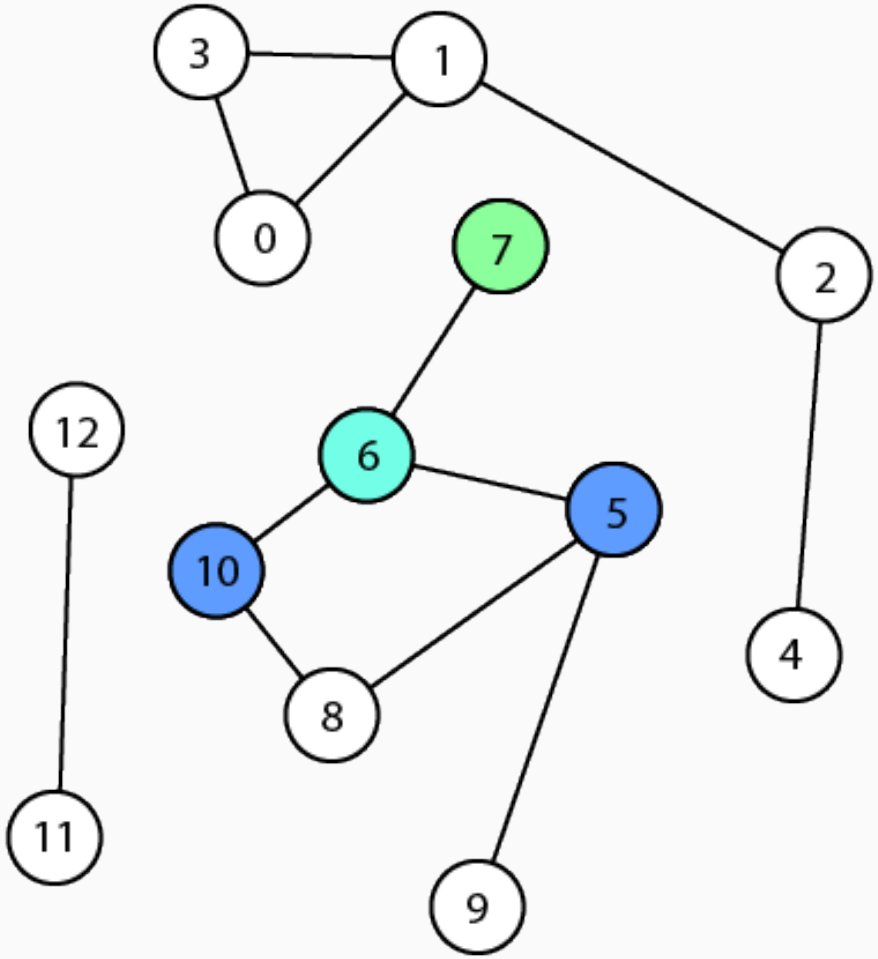
Generations:

[7]

[6]

[10, 5]

[8, 9]



Generations / Queue

Generations:

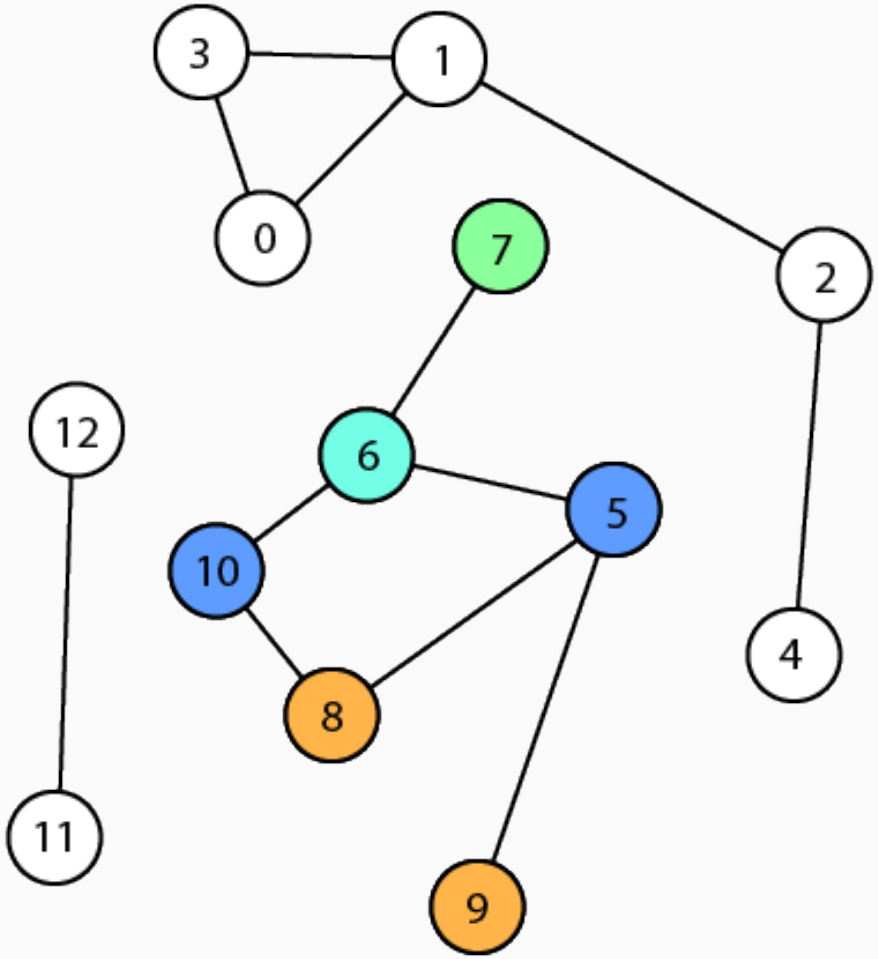
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations:

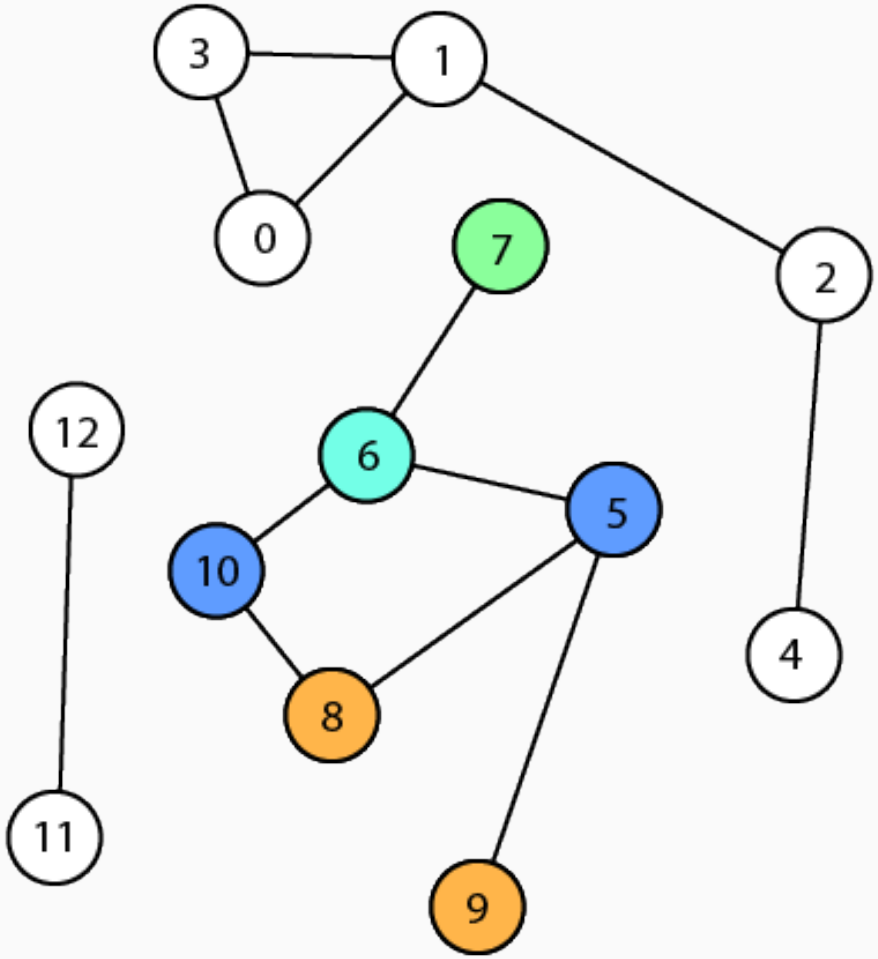
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations:

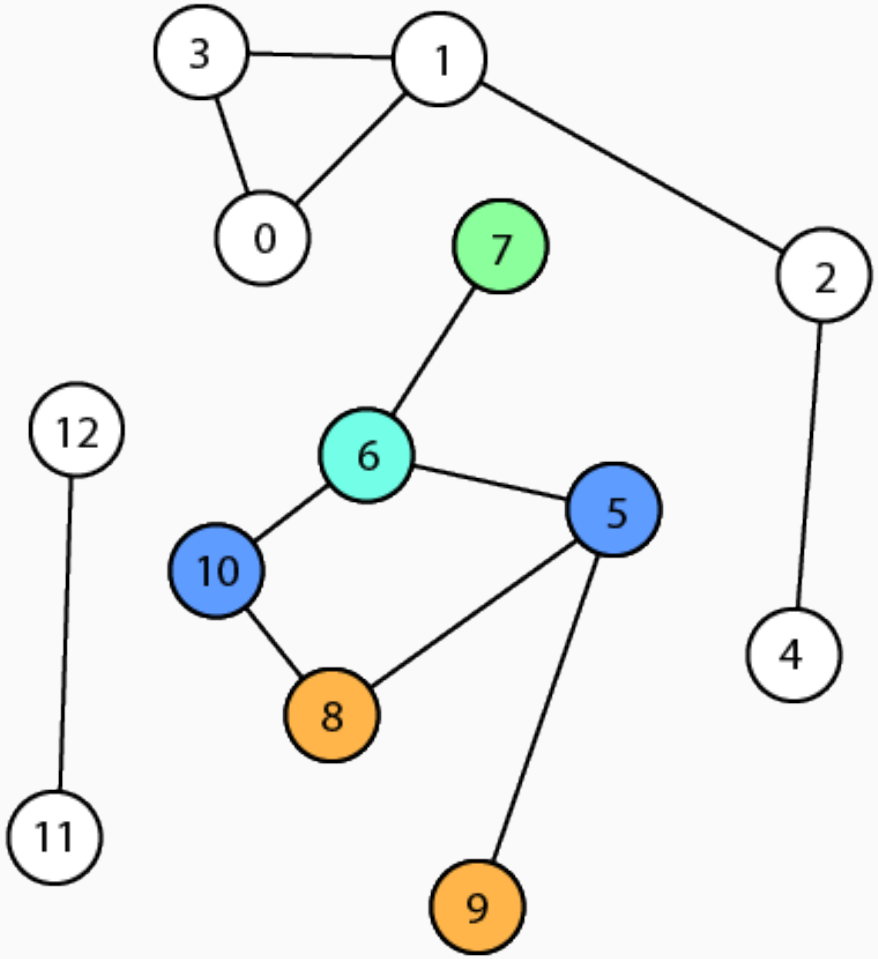
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations:

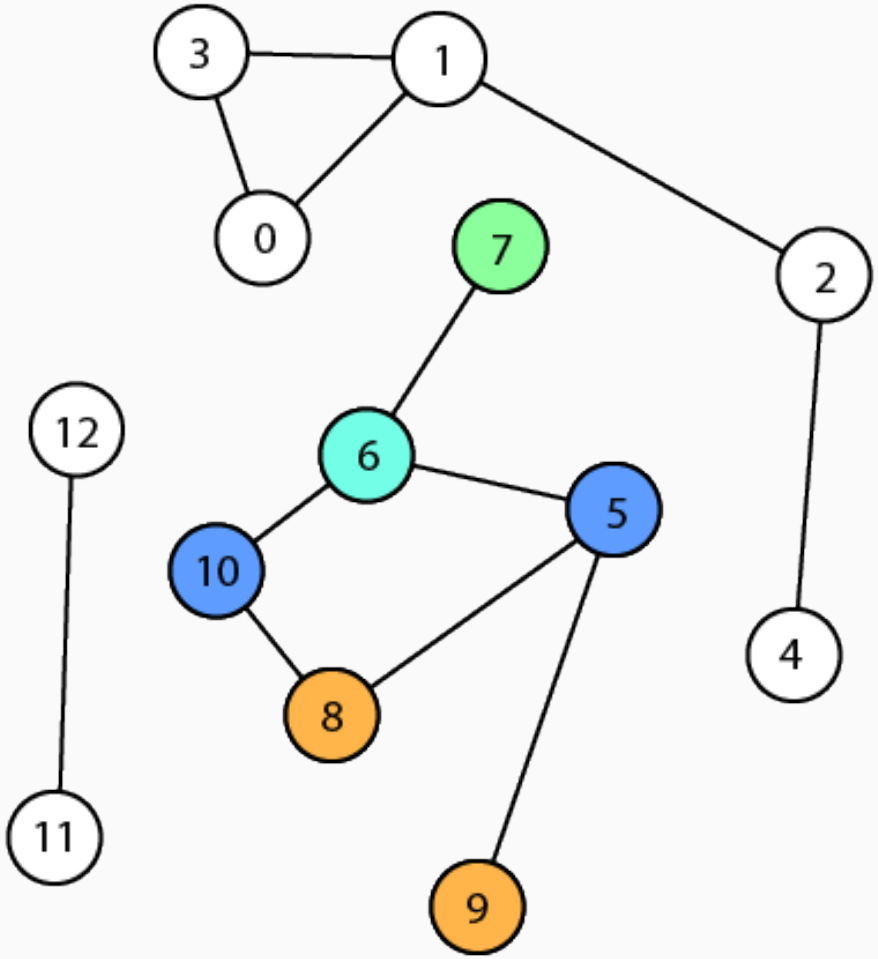
[7]

[6]

[10, 5]

[8, 9]

[ ]



Generations / Queue

Generations

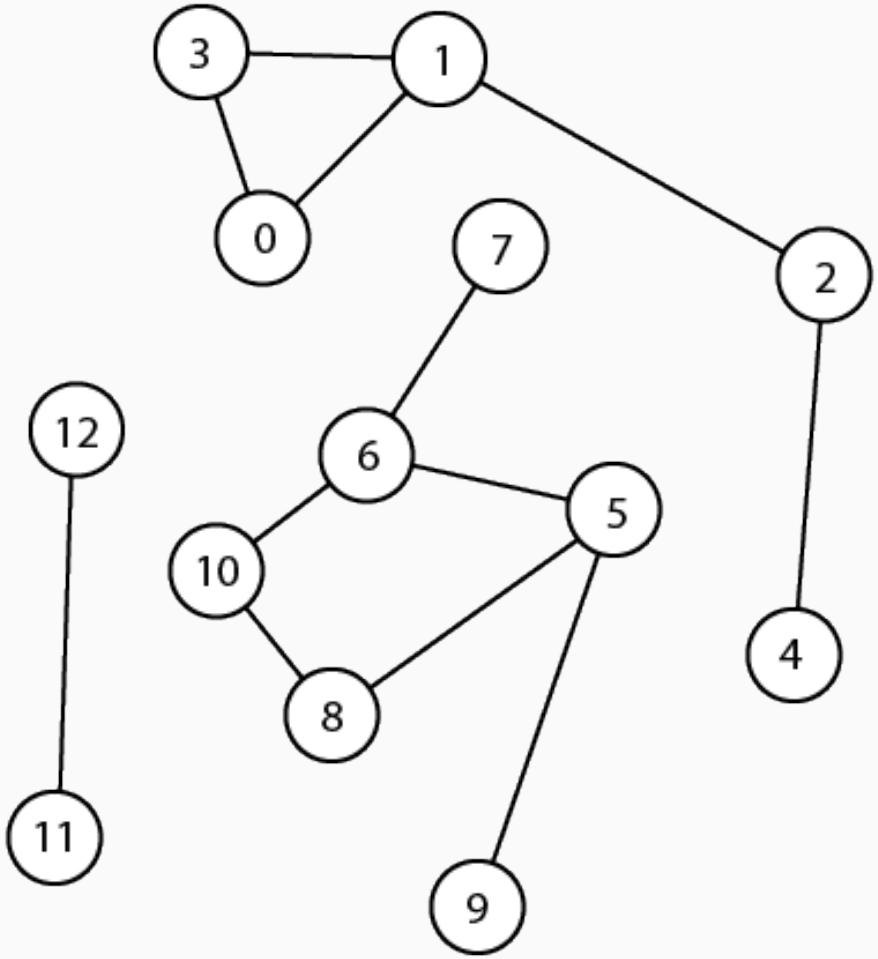
[7]
[6]
[10, 5]
[8, 9]

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]



Interlude: Queue

- Datastructure for FIFO
- First In, First Out (like queue in mensa)
- Can push elements at end
- Can pop elements from front

```
1  queue<int> q;
2  q.push(7);
3  q.push(4);
4  int seven = q.front();
5  q.pop();
6  q.push(5);
7  q.pop();
8  int one = q.size();
9  int five = q.front();
```

BFS Implementation

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

```
1  vector<vector<int>> graph(n);
2  vector<int> vis(n, 0);
3  queue<int> q;
```

BFS Implementation

- Process start node

```
1  ...
2  q.push(start);
3  vis[start] = 1;
```

BFS Implementation

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

```
1   ...
2   while (!q.empty()) {
3       int v = q.front();
4       q.pop();
5       for (int w : graph[v]) {
6           if (vis[w] == 0) {
7               vis[w] = 1;
8               q.push(w);
9           }
10      }
11  }
```

BFS Implementation

- Return visited nodes

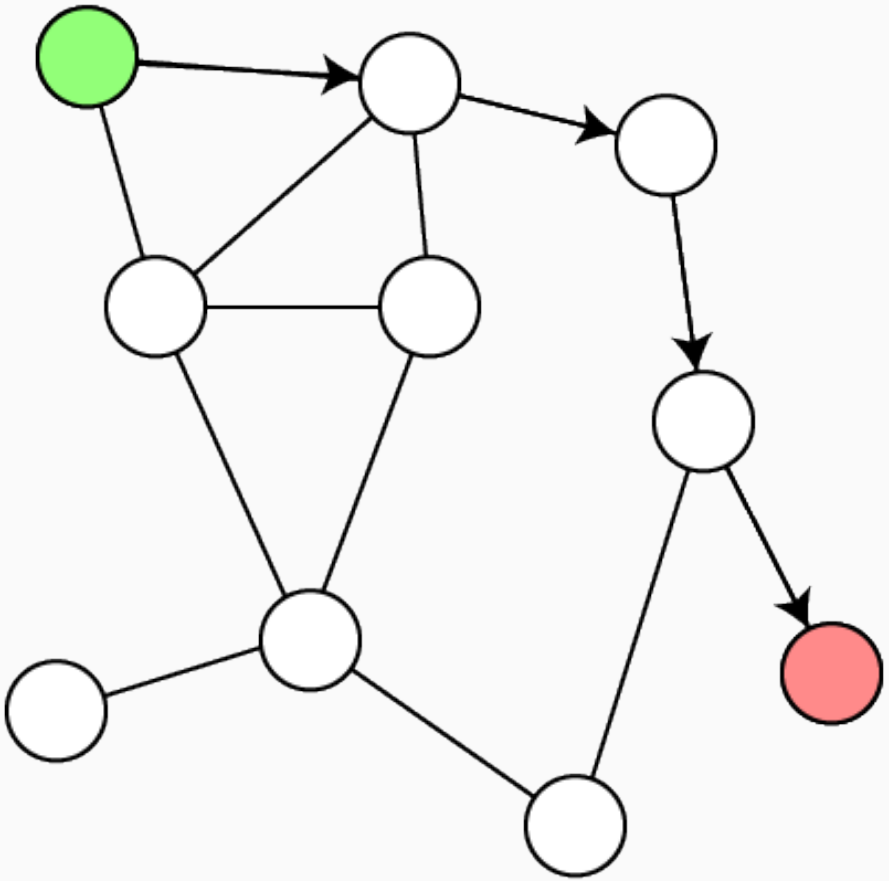
```
1    ...
2    return vis;
```

BFS Implementation

```
1   vector<vector<int>> graph(n);
2   vector<int> vis(n, 0);
3   queue<int> q;
4   q.push(start);
5   vis[start] = 1;
6   while (!q.empty()) {
7       int v = q.front();
8       q.pop();
9       for (int w : graph[v]) {
10          if (vis[w] == 0) {
11              vis[w] = 1;
12              q.push(w);
13          }
14      }
15  }
```

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

```
1  ...
2  vector<int> dist(n, -1);
3  ...
```

Shortest Distance Implementation

- Store distance

```
1  ...
2  int d = dist[v];
3  ...
4  dist[w] = d + 1;
5  q.push(w);
6  ...
```

Shortest Distance Implementation

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

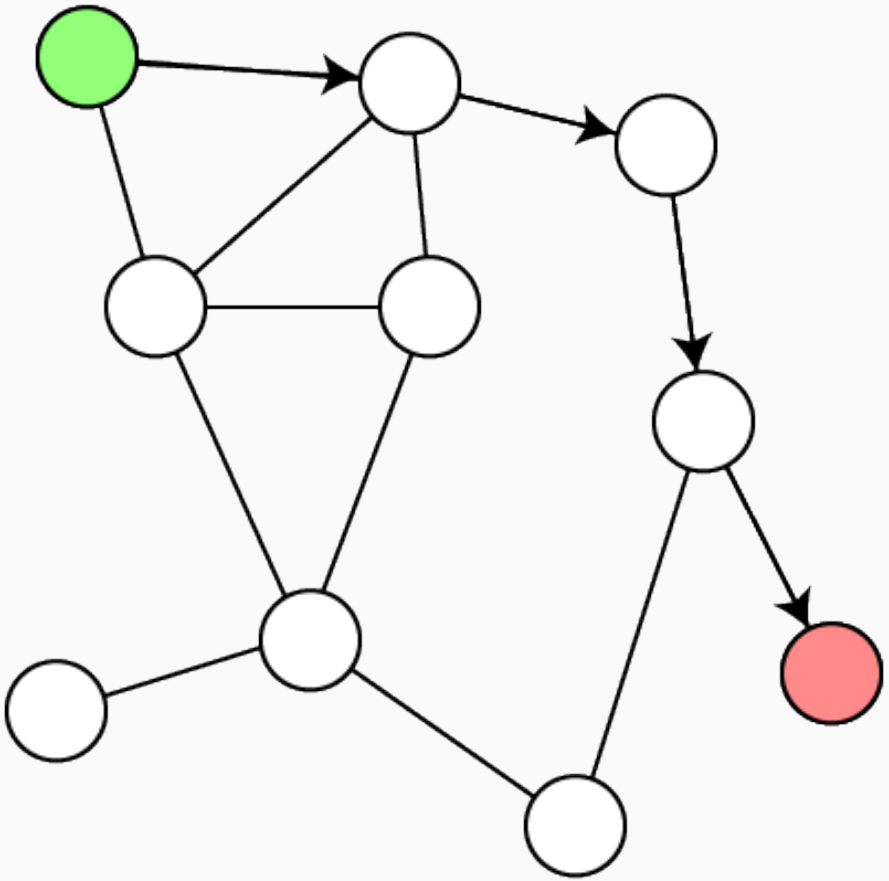
```
1    ...
2    return dist[target];
```

Shortest Distance Implementation

```
1   vector<vector<int>> graph(n);
2   vector<int> dist(n, -1);
3   queue<int> q;
4   q.push(start);
5   dist[start] = 0;
6   while (!q.empty()) {
7       int v = q.front();
8       q.pop();
9       int d = dist[v];
10      for (int w : graph[v]) {
11          if (dist[w] == -1) {
12              dist[w] = d + 1;
13              q.push(w);
14          }
15      }
16  }
17  return dist[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

```
1  ...
2  vector<int> par(n, -1);
3  ...
```

Shortest Path Implementation

- Store parent

```
1  ...
2  par[w] = v;
3  q.push(w);
4  ...
```

Shortest Path Implementation

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

```
1  vector<int> path;
2  if (par[target] == -1) {
3      return path;
4  }
5  int current = target;
6  while (current != par[current]) {
7      path.push_back(current);
8      current = par[current];
9  }
10 path.push_back(start);
11 return vector<int>(path.rbegin(), path.rend());
```

Shortest Path Implementation

```
1   vector<vector<int>> graph(n);
2   vector<int> par(n, -1);
3   queue<int> q;
4   q.push(start);
5   par[start] = start;
6   while (!q.empty()) {
7       int v = q.front();
8       q.pop();
9       for (int w : graph[v]) {
10          if (par[w] == -1) {
11              par[w] = v;
12              q.push(w);
13          }
14      }
15  }
16
17  vector<int> path;
18  if (par[target] == -1) {
19      return path;
20  }
21  int current = target;
22  while (current != par[current]) {
23      path.push_back(current);
24      current = par[current];
25  }
26  path.push_back(start);
27  return vector<int>(path.rbegin(), path.rend());
```

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