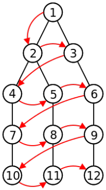
Graphs: Breadth-First Search (BFS)

L.EIC

Algorithms and Data Structures

2025/2026



P Ribeiro, AP Tomas

Breadth-First Search - BFS

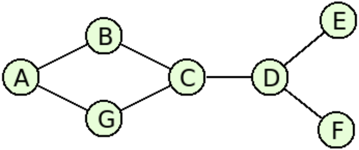
- A **breadth-first search** (**BFS**) is very similar to a DFS. Essentially, it only changes the **order** in which the nodes are visited!
- Instead of using recursion (and its *stack*), we will explicitly keep a **queue** of unvisited nodes (q)

Backbone of a BFS - time: $\mathcal{O}(|V| + |E|)$ (adj. list)

bfs(node v):
    $q \leftarrow \emptyset$ /* queue of unvisited nodes */
    $q.enqueue(v)$
    mark v as visited /* all other nodes should initially be unvisited */
    While $q \neq \emptyset$ /* while there are still unprocessed nodes */
        $u \leftarrow q.dequeue()$ /* remove first element of q */
        For all neighbors w of u **do**
            If w has not yet been visited **then** /* new node! */
                $q.enqueue(w)$
                mark w as visited

Breadth-First Search - BFS

- Here is a graph and an illustration of a BFS starting on node A:



1. Initially we have $q = \{A\}$
2. We visit and remove A, adding its unvisited neighbors ($q = \{B, G\}$)
3. We visit and remove B, adding its unvisited neighbors ($q = \{G, C\}$)
4. We visit and remove G, adding its unvisited neighbors ($q = \{C\}$)
5. We visit and remove C, adding its unvisited neighbors ($q = \{D\}$)
6. We visit and remove D, adding its unvisited neighbors ($q = \{E, F\}$)
7. We visit and remove E, adding its unvisited neighbors ($q = \{F\}$)
8. We visit and remove F, adding its unvisited neighbors ($q = \{\}$)
9. q empty, we finished our BFS

BFS: Implementation

- Let's see a possible implementation with some *livecoding*

```
void bfs(int v) {
  // initialize all nodes as unvisited
  for (int v=1; v<=n; v++) nodes[v].visited = false;
  queue<int> q; // queue of unvisited nodes
  q.push(v);
  nodes[v].visited = true;
  while (!q.empty()) { // while there are still unprocessed nodes
    int u = q.front(); q.pop(); // remove first element of q
    cout << u << " "; // show node order
    for (auto e : nodes[u].adj) {
      int w = e.dest;
      if (!nodes[w].visited) { // new node!
        q.push(w);
        nodes[w].visited = true;
      }
    }
  }
}
```

BFS: Implementation

- Example execution (assuming graph g was already created)

```
g.bfs(1);
```

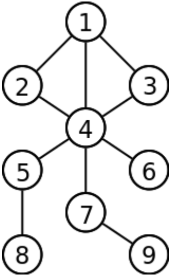
```
1 2 3 4 5 6 7 8 9
```

```
g.bfs(9);
```

```
9 7 4 1 2 3 5 6 8
```

```
g.bfs(5);
```

```
5 4 8 1 2 3 6 7 9
```



BFS: Computing distances

- Almost everything than can be done with DFS can also be done with BFS!
- An important difference is that with BFS we visit the nodes in **increasing order of distance** (in terms of number of edges) to the initial node!
- In this way, BFS can be used to compute **shortest distances** between nodes on an **unweighted graph** (with or without direction).
- Let's see what really changes in the code

BFS: Computing distances

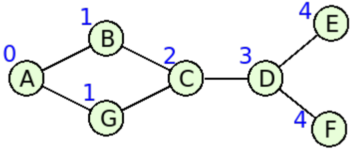
- In red the lines that were added.
  node.distance stores the distance to *node*.

BFS - Computing distances

bfs(node v**)**:
  $q \leftarrow \emptyset$ /* Queue of unvisited nodes */
  $q.enqueue(v)$
  $v.distance \leftarrow 0$ /* distance from v to itself it's zero */
  mark v as visited /* all other nodes should initially be unvisited */
  While $q \neq \emptyset$ /* while there are still unprocessed nodes */
    $u \leftarrow q.dequeue()$ /* remove first element of q */
    For all neighbors w of u **do**
      If w has not yet been visited **then** /* new node */
        $q.enqueue(w)$
        mark w as visited
        $w.distance \leftarrow u.distance + 1$

BFS example with distances

- Here is an illustration of a BFS (with distances) starting on node A:



1. Initially we have $q = \{A\}$ and $A.distance = 0$
2. We visit A, adding its neighbors ($q = \{B, G\}$; $B.distance = G.distance = 1$)
3. We visit B, adding its neighbors ($q = \{G, C\}$; $C.distance = 2$)
4. We visit G, adding its neighbors ($q = \{C\}$)
5. We visit C, adding its neighbors ($q = \{D\}$; $D.distance = 3$)
6. We visit D, adding its neighbors ($q = \{E, F\}$; $E.distance = F.distance = 4$)
7. We visit E, adding its neighbors (($q = \{F\}$)
8. We visit F, adding its neighbors ($q = \{\}$)
9. q empty, we finished our BFS

BFS: Computing paths

- What if we want to **report the path** to reach a node with distance?
- We could simply store for each node v its "predecessor" ($pred$), or the node that added v to the queue
- In this way, a path can be reconstructed by following the predecessor of the target note to get the path in inverse direction:

  path to node v: $\ldots \rightarrow pred(pred(pred(v))) \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v$
- Let's see what would change in the code

BFS: Computing paths

- Added lines in red. *node.pred* stores its predecessor.

BFS - Computing distances

bfs(node v):
  $q \leftarrow \emptyset$ /* Queue of unvisited nodes */
  $q.enqueue(v)$
  $v.distance \leftarrow 0$
  $v.pred \leftarrow v$ /* source node - no predecessor */
  mark v as visited /* all other nodes should initially be unvisited */
  While $q \neq \emptyset$ /* while there are still unprocessed nodes */
    $u \leftarrow q.dequeue()$ /* remove first element of q */
    For all neighbors w of u **do**
      If w has not yet been visited **then** /* new node */
        $q.enqueue(w)$
        mark w as visited
        $w.distance \leftarrow u.distance + 1$
        $w.pred \leftarrow u$

BFS: More applications (different graph types)

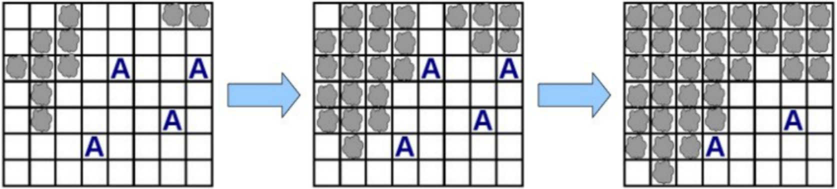
- BFS can be applied in many different graph types
- Consider for instance that you want to know the **minimum distance between** points **A** and **B** on a 2D maze:

```
########                            ########
#A.....#                            #A12345#
####.###          --->              ####4###
#B.....#    BFS starting in A       #876567#
########                            ########
```

  - A node is a cell (x, y)
  - Neighbors are $(x+1, y)$, $(x-1, y)$, $(x, y+1)$ e $(x, y-1)$
  - Everything else in the BFS is the same! (time: $\mathcal{O}(rows \times cols)$)
  - To store on the queue we need to represent a pair of coordinates (in C++ this could for instance be: `queue<pair<int, int>> q`).

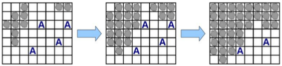
BFS: More applications (multiple sources)

- Let's see an old problem from a qualification phase of the National Olimpiads in Informatics (ONI'2010)
- The problem was inspired by **Eyjafjallajökull volcano** eruption, whose ash cloud completely disrupted airplane traveling in Europe.
- Imagine that the **ash cloud** is given as a matrix and that in each time the cloud expands by one cell both horizontally and vertically. The `A`'s are the airports.



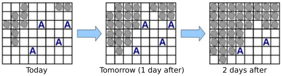
Today   Tomorrow (1 day after)   2 days after

BFS: More applications (multiple sources)



- The problem asked for:
  - How many days until the **first airport** is covered by ashes
  - How many days until **all airports** are covered by ashes
- Let $dist(A_i)$ be the distance of airport i up to any ash cloud
- The problems asks for the smallest $dist(A_i)$ and the largest $dist(A_i)$
- One option could be to make one BFS per airport
  $\mathcal{O}(num\_aiports \times rows \times cols)$
- Another option could be to make one BFS per ash cell
  $\mathcal{O}(num\_ashes \times rows \times cols)$
- How to do better, using only one BFS?

BFS: More applications (multiple sources)



- Idea: initialize the BFS queue with all ash positions!
- Everything else remains the same.

```
...#...        ..1#1..       .21#12.       321#123       321#123
..##...        .1##1..       21##12.       21##123       21##123
.####..   ->   1####1.  ->   1####12  ->   1####12  ->   1####12
.......        11111..       111112.       1111123       1111123
##.....        ##1....       ##122..       ##1223.       ##12234
```

- The computed distances are exactly what we need
- Each cell will only be traversed once! $\mathcal{O}(rows \times cols)$

BFS: More applications (implicit graphs and game search)

- Let's see a final problem where there is no "explicit" graph *[original problem from IOI'1996]*
- Consider the following puzzle (almost like a 2D version of Rubik's cube)
  - The initial board position is:

1	2	3	4
8	7	6	5

  - In each turn you can make of these 3 movement types:
    - **Movement A**: swap upper and lower rows

8	7	6	5
1	2	3	4

    - **Movement B**: shift the rectangle to the right

4	1	2	3
5	8	7	6

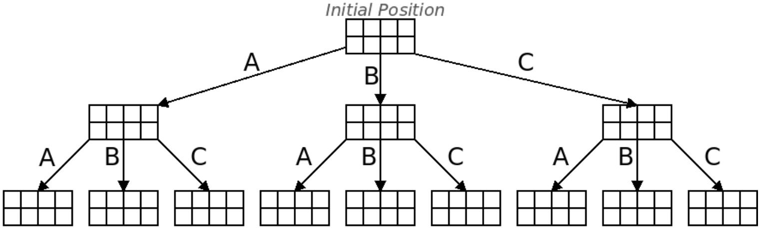
    - **Movement C**: clockwise rotation of the 4 inner cells

1	7	2	4
8	6	3	5

  - **How many turns** do we need to reach a certain position?

BFS: More applications (implicit graphs and game search)

- This can be solved with... **BFS**!
- The **initial node** is... the starting position.
- The **adjacent nodes** are... the positions you can reach using movements A, B or C.



- When we reach the desired position... we known the **shortest distance** (number of movements) to reach it!
- The "hardest" part is... knowing how to **represent and manipulate the board positions** ! :)

Graph Search - Wrap Up

- One of the most fundamental tasks with graphs is **graph search** (or **graph traversal**): passing through all the nodes using the links
- There are two fundamental graph search methodologies that vary on **order in which they traverse the nodes**:
  - **Depth-First Search - DFS**
    Traverse the entire subgraph connected to a neighbor before entering the next neighbor node
  - **Breadth-First Search - BFS**
    Traverse the nodes by increasing distance of number of links to reach them
- Besides the slides and theses two classes presentations here are two other **visualizations** of these traversal algorithms:
  - VisuAlgo: Graph Traversal (DFS/BFS)
  - David Galles' visualizations: DFS and BFS