#### < **Graph Traversal**

* Objective: Visit every vertex and every edge exactly once
* Purpose: Search for interesting substructures in the graph
* We’ve done it on trees, but it was easier

|  |  |
| --- | --- |
| **Trees** | **Graphs** |
| 1. **Ordered** → we always go from parents to children. 2. **Obvious** **start** → we start at the root node. 3. **Notion of completeness** → we are done when we reach leaf nodes. | 1. **Unordered** → no notion of children nodes, just neighbours. 2. **No** obvious start → we can start anywhere. 3. **No** notion of completeness → we need to know when we have visited all nodes. |

#### **BFS Algorithm:**

1. Add the starting point
2. While the queue is not empty
   1. Dequeue **v**
   2. For all of the unlabelededges adjacent to **v**
      * If an adjacent edge “discovers” a new vertex **t**:
        + Label the edge a “discovery edge”
        + Enqueue **t**, update the information of **t** (distance = dist(**v**) + 1, predecessor = **v**)
      * If an adjacent edge is between two visited vertices
        + Label the edge a “cross edge”

* **Example**: see previous lecture notes / video
* 1
* 2
* 3
* **The code**

|  |  |
| --- | --- |
| 4  5  6  7  8  9  10  11  12  13 | BFS(G):  Input: Graph, G  Output: A labeling of the edges on  G as discovery and cross edges  foreach (Vertex v : G.vertices()):  setLabel(v, UNEXPLORED)  foreach (Edge e : G.edges()):  setLabel(e, UNEXPLORED)  foreach (Vertex v : G.vertices()):  if getLabel(v) == UNEXPLORED:  BFS(G, v)  //components++; |

C

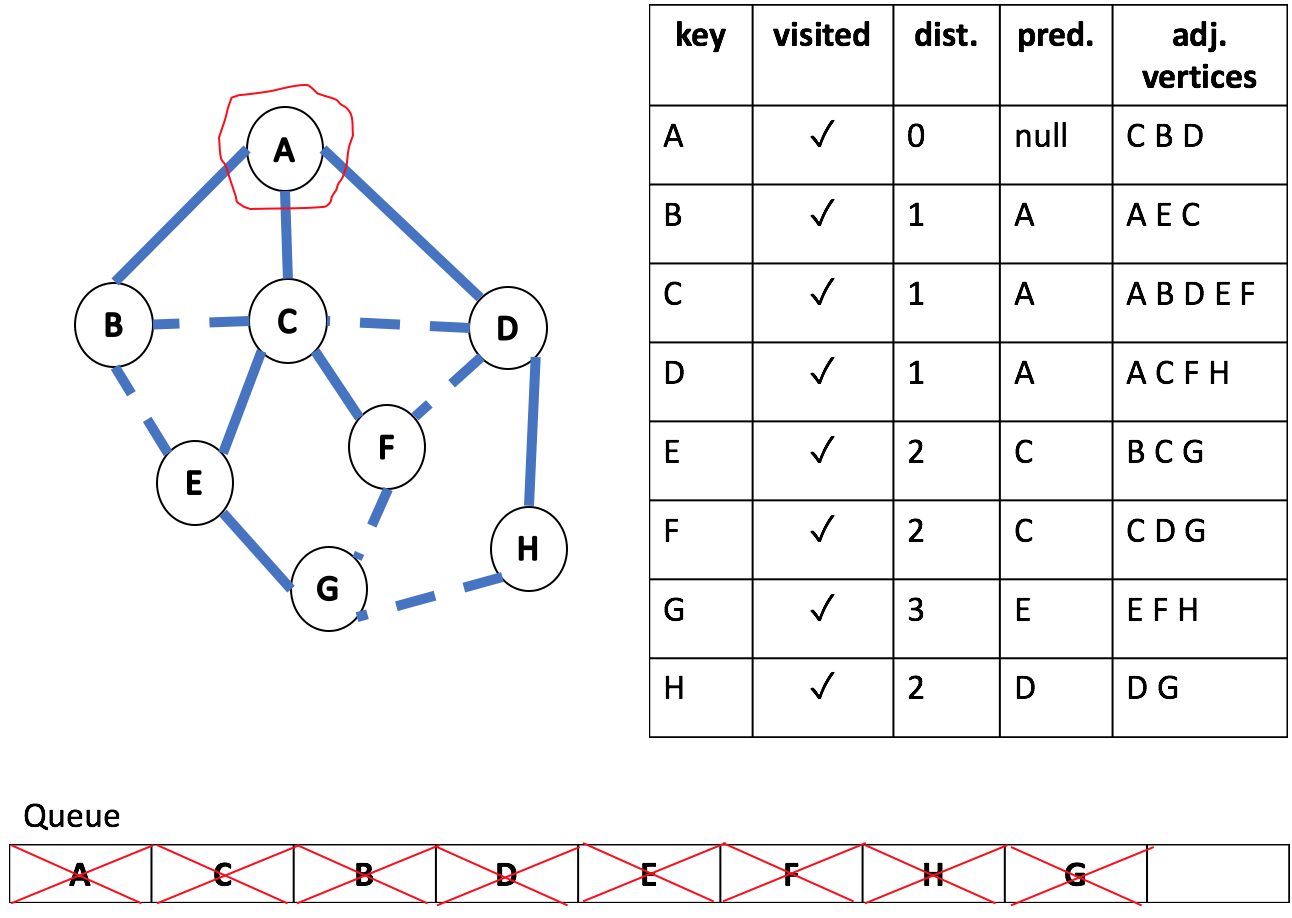
|  |  |
| --- | --- |
| 14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | BFS(G, v):  Queue q  setLabel(v, VISITED)  q.enqueue(v)  while !q.empty():  v = q.dequeue()  foreach (Vertex w : G.adjacent(v)):  if getLabel(w) == UNEXPLORED:  setLabel(v, w, DISCOVERY)  setLabel(w, VISITED)  q.enqueue(w)  lse  if getLabel(v, w) == UNEXPLORED:  setLabel( v, w, CROSS)  // cycleExists = true; |

* Use cases and functionality:
  + Does this code work on a disjoint graph (2 or more separate pieces)?
    - Yes, since line 10 goes through every vertex, regardless of connectedness
  + How do we use the traversal to count the number of components?
    - Every BFS run indicates a connected component, so we can add a counter after a call of BFS (line 13).
  + Can our implementation detect a cycle?
    - Yes, a cross edge indicates a cycle (line 28).

#### **Running time:**

* + Expect: visit every edge and vertex, so O(n+m)
  + Looking at specific parts of the code:
    - Second part of the code:
      * Line 19:
      * Line 21:
      * Lines 22-27:
      * This whole chunk is
      * is not very informative, but we know that we will have
    - First part of the code:
      * Lines 6-7:
      * Lines 8-9:
      * Lines 10-12:
  + Total running time is .
  + This is optimal running time because we know we have to visit every edge and vertex, therefore we cannot do better than .
* BFS doesn’t give a unique solution, but the properties are guaranteed.

#### **BFS Observations**



* What is the shortest path from A to H?
  + path along discovery edges: A->D->H
* What is the shortest path from E to H?
  + Actual: E->G->H
  + Is not obvious in the BFS result
  + **BFS only finds the shortest paths from the start**
    - single source shortest path
* How does a cross edge related to **dist**?
  + Cross edge will never change the **dist** more than 1.
  + BFS keeps things local: every edge increase/decrease distance by 1.
* What structure is made from discovery edges?
  + A tree (forest, if the graph is not connected) rooted at the start
  + Further, it’s a spanning tree (forest): it connects all vertices in the graph

#### **DFS**

* Idea:
  + similar idea with BFS, but we can use either
    - A stack
    - A recursion
  + Recursive algorithm: we visit a vertex **v**

1. Check all adjacent vertices of **v**
   1. if it’s not been discovered, label the edge “discovery edge”. Visit the new vertex
   2. label the edge that leads us to a already discovered vertex “back edge”
      1. since it *usually* brings us to a closer vertex
      2. the distance difference is unbounded
   * Observations
     + The discovery edges make a spanning tree
     + **d** does not find the shortest path
     + the benefit is: it discovers new vertices very quickly

* The code: use system stack as our workstack (recursion)