Graphs: Breadth-First Search

February 11, 2011
Recall what a graph looks like.
This one is undirected and unweighted.

An Example Graph

Graphs: Breadth-First Search
What is Breadth-First Search?

Breadth-first search (BFS) is an algorithm for systematically visiting all nodes of a graph that are reachable from a given “source” node $s$.

It also computes the minimum number of steps needed to get from $s$ to each other node.

Applications:

- Determine if there a path from node $s$ to node $t$.
- Find the portion of the graph reachable from $s$. (Computing connected components.)
- Find the shortest path from $s$ to $t$ (in an unweighted graph).
- Find a spanning tree of an undirected graph.
- And more...
What is Breadth-First Search?

Breadth-first search (BFS) is an algorithm for systematically visiting all nodes of a graph that are reachable from a given “source” node $s$.

It also computes the minimum number of steps needed to get from $s$ to each other node.

Applications:

- Determine if there a path from node $s$ to node $t$.
- Find the portion of the graph reachable from $s$. (Computing connected components.)
- Find the shortest path from $s$ to $t$ (in an unweighted graph).
- Find a spanning tree of an undirected graph.
- And more...
Breadth-first search (BFS) is an algorithm for systematically visiting all nodes of a graph that are reachable from a given “source” node \( s \).

It also computes the minimum number of steps needed to get from \( s \) to each other node.

Applications:

- Determine if there a path from node \( s \) to node \( t \).
- Find the portion of the graph reachable from \( s \).
  (Computing connected components.)
- Find the shortest path from \( s \) to \( t \) (in an unweighted graph).
- Find a spanning tree of an undirected graph.
- And more...
What is Breadth-First Search?

Breadth-first search (BFS) is an algorithm for systematically visiting all nodes of a graph that are reachable from a given “source” node $s$.

It also computes the minimum number of steps needed to get from $s$ to each other node.

Applications:

- Determine if there a path from node $s$ to node $t$.
- Find the portion of the graph reachable from $s$. (Computing connected components.)
- Find the shortest path from $s$ to $t$ (in an unweighted graph).
- Find a spanning tree of an undirected graph.
- And more...
Breadth-first search (BFS) is an algorithm for systematically visiting all nodes of a graph that are reachable from a given “source” node $s$.

It also computes the minimum number of steps needed to get from $s$ to each other node.

Applications:

- Determine if there a path from node $s$ to node $t$.
- Find the portion of the graph reachable from $s$. (Computing connected components.)
- Find the shortest path from $s$ to $t$ (in an unweighted graph).
- Find a spanning tree of an undirected graph.
- And more...
Breadth-first search (BFS) is an algorithm for systematically visiting all nodes of a graph that are reachable from a given “source” node $s$.

It also computes the minimum number of steps needed to get from $s$ to each other node.

Applications:

- Determine if there a path from node $s$ to node $t$.
- Find the portion of the graph reachable from $s$.
  (Computing connected components.)
- Find the shortest path from $s$ to $t$ (in an unweighted graph).
- Find a spanning tree of an undirected graph.
- And more...
Maintain two sets of nodes:
- \( T \) stores nodes that have already been visited, and
- \( Q \) stores nodes that we would like to visit in the future.

Initially, \( T = \{\} \) and \( Q = \{s\} \).

Repeatedly choose a node \( u \) from \( Q \) to visit next. Move \( u \) to \( T \).
When we visit \( u \), add \( u \)'s unvisited neighbours to \( Q \).
Pseudocode for Search

\[ Q = \{s\} \]
\[ T = \{\} \]

while \( Q \) is not empty
    remove a node \( u \) from \( Q \)
    add \( u \) to \( T \)
    for each edge \( u \rightarrow v \) in the graph
        if \( v \) is not already in \( T \), add \( v \) to \( Q \)
    end for
end while

If we implement \( Q \) as a FIFO queue, then this algorithm is breadth-first search.
Pseudocode for Search

\[ Q = \{s\} \]
\[ T = \{\} \]

while \( Q \) is not empty
    remove a node \( u \) from \( Q \)
    add \( u \) to \( T \)
    for each edge \( u \rightarrow v \) in the graph
        if \( v \) is not already in \( T \), add \( v \) to \( Q \)
    end for
end while

If we implement \( Q \) as a FIFO queue, then this algorithm is breadth-first search.
Graphs: Breadth-First Search

\[
T = \{\}
\]

\[
Q = \{A\}
\]
Example

Graphs: Breadth-First Search

\[ T = \{ A \} \]
\[ Q = \{ B, C \} \]
Example

Graphs: Breadth-First Search

\[ T = \{A, B\} \]
\[ Q = \{C, D, E\} \]
Example

Graphs: Breadth-First Search

\[ T = \{A, B, C\} \]
\[ Q = \{D, E, F\} \]
$T = \{A, B, C, D\}$

$Q = \{E, F\}$
Graphs: Breadth-First Search

$T = \{A, B, C, D, E\}$

$Q = \{F\}$
Example

Graphs: Breadth-First Search

\[ T = \{ A, B, C, D, E, F \} \]
\[ Q = \{ \} \]
When a node $v$ is added to $Q$ because of edge $u \rightarrow v$, $v$ stores pointer to $u$.

This forms a tree of shortest paths towards the source of the BFS.

This also makes it easy to compute distances from source to every other node.
When a node $v$ is added to $Q$ because of edge $u \rightarrow v$, $v$ stores pointer to $u$.

This forms a tree of shortest paths towards the source of the BFS.

This also makes it easy to compute distances from source to every other node.
list[i] is adjacency list of node i.
distance[i] is computed distance from s to i (initially −1).
parent[i] is parent of i in shortest path tree (initially −1).

Queue<Integer> Q = new LinkedList<Integer>();
distance[s]=0;
Q.add(s);
while (!Q.isEmpty()) {
    int u = Q.remove();
    for (int v : list[u]) { // for each edge u -> v
        if (distance[v] == -1) { // v not in Q or T
            distance[v] = distance[u] + 1;
            Q.add(v);
            parent[v] = u;
        }
    }
}

Graphs: Breadth-First Search
Remarks on implementation

- See last week’s slides for how adjacency lists are created.
- Note that \( T \) is not represented explicitly.
- The add and remove operations of the LinkedList class implement a FIFO queue.
- Exactly the same code works for directed and undirected graphs.
BFS can be used for computing the shortest paths from $s$ to other nodes.

Two ways to print the shortest path from $s$ to $t$ (if it exists):

- Follow parent pointers from $t$ to $s$ and then reverse the path computed.
- Use recursion:
  ```
  printPathTo(t)
  if $t \neq s$ then printPathTo(parent(t))
  print $t$
  ```